André Daniel Marques Simões Gonçalves

Licenciado em Engenharia Informática

MixCloud
Middleware para Replicação e Integração de Repositórios Chave/Valor

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador :	Doutor Víctor Manuel Alves Duarte, Prof. Auxiliar, Universidade Nova de Lisboa
Co-orientador :	Doutor Nuno Manuel Ribeiro Preguiça, Prof. Auxiliar, Universidade Nova de Lisboa

	Júri:
Presidente:	Doutor Adriano Martins Lopes
Arguente:	Doutor João Coelho Garcia
Vogal:	Doutor Vítor Manuel Alves Duarte

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2011

MixCloud
Middleware para Replicação e Integração de Repositórios Chave/Valor

À minha Família, ao Scooby e aos meus Amigos.

Agradecimentos

O meu percurso académico, até este ponto, tem sido recheado de conhecimento e muita alegria por aprender coisas novas. Como a minha querida mãe diz, tudo começou no infantário desde o três anos. Feitas as contas são 21 anos a estudar. Foi nos meus pais que encontrei, no início, a inspiração para crescer e conhecer o mundo à minha volta. Aprendi com a minha mãe as primeiras letras e a escrever o meu nome, as lenga-lengas, as músicas e a tabuada. Com o meu pai aprendi a ser responsável, trabalhador e dar valor às pessoas que me eram próximas. Sempre e junto deles fui avançando e mostrando o meu valor enquanto pessoa e que poderia aspirar a grandes feitos no futuro. Este foi o primeiro passo, graduar-me e partir para o mundo do trabalho. A vocês dois, as pessoas que eu mais amo no mundo, o meu muito obrigado por me terem apoiado e ajudado a avançar com muito amor e carinho.

Ao meu querido irmão e cunhada, por todas as vezes que me receberam de braços abertos e me dão força para continuar no meu caminho. Especialmente ao meu irmão, que sempre mostrou orgulho em mim e confiou nas minhas capacidades. Também às minhas sobrinhas, a alegria da minha vida, por me mostrarem que vida pode ser simples e aproveitada sempre com um sorriso nos lábios. Ao resto da minha família que sempre esteve do meu lado e se preocupam com o meu futuro. Por estarem interessados no que estudo e nos projectos em que estou inserido. Por me perguntarem sem parar : "então quando é que acabas os estudos?".

Queria agradecer ao Departamento de Informática (DI) da FCT-UNL por me ter acolhido durante estes 6 anos de formação. Agradeço igualmente à Fundação para as Ciências e Tecnologia pela bolsa de investigação concedida.

Aos meus orientadores, Victor Manuel Alves Duarte e Nuno Manuel Preguiça, não só pela orientação, mas também pela paciência e pelas discussões produtivas que tivemos. Obrigado por me terem ajudado a realizar este trabalho e pela oportunidade que me deram.

A todos os colegas que acompanharam desde que cheguei à FCT, especialmente aqueles que partilhavam o gabinete 254 comigo durante o período que realizei a dissertação. As nossas discussões sobre os trabalhos e a sociedade em geral foram essenciais para

melhorar este trabalho.

O meu também especial agradecimento à minha professora da primária Isaura, a pessoa que me passou os conhecimentos básicos e fomentou o meu gosto pelo estudo. Sei que já não está por cá, mas quero dizer o quanto a admirava e importância que tinha para mim. Espero que esteja atenta e veja o quão bem os seus alunos têm saído.

Quero agradecer aos meus amigos, aqueles que sem a sua existência nada disto seria possível. Relembro as longas conversas que tínhamos, na companhia de café, sobre o nossas alegrias e frustrações e de como ser estudante era difícil. O vosso apoio e a força que me transmitiram foram fundamentais para o meu sucesso. Aprendi imenso com vocês! Nunca vos esquecerei: Bruno Letras (Zé), Sílvia, Mafs, Ana (Pseudo-Prima), Mário, Sara, Luís, Paula, Sofia, Miguel, Cátia, Joana e ao meu colega Ricardo.

Por fim ao meu colega de casa Raimundo, por ser tão meu amigo e ajudar-me todos os dias a ser uma pessoa melhor. Foste o meu mentor e deste um contributo imenso neste trabalho, pelo qual estou eternamente grato. Todas as discussões iluminadas sobre as minhas dúvidas científicas foram essências para compreender melhor o meu trabalho. Agradeço também por todas as vezes que me tiras-te de casa para passear e aproveitar o outro lado da vida. You are my inspiration!

Resumo

Nos últimos anos têm sido desenvolvidos um grande número de repositórios chave/-valor. Em particular, estes repositórios são utilizados em muitas das aplicações de *cloud computing*, garantindo o seu correcto funcionamento, elevada disponibilidade, desempenho e escalabilidade. As diferentes soluções fornecem desempenhos variados para diferentes operações. Por outro lado, as interfaces de programação que apresentam diferem entre si, influenciando o desenho das aplicações e comprometendo estas a um repositório específico. Neste domínio existem sistemas que não oferecem garantias de replicação interna, com impacto significativo sobre a disponibilidade dos dados e desempenho. Nesta dissertação apresentar-se o sistema MixCloud, que permite fornecer um serviço de repositório chave/valor replicado. Para tal, integra vários repositórios chave/valor existentes, procurando combinar as melhores características de cada um deles. A solução oferece uma interface independente dos sistemas usados. O mecanismo de replicação adopta um modelo de consistência final dos dados, baseado na utilização da transformação de operações. Desse modo, o resultado de cada operação é retornado assim que o repositório mais rápido completa a sua execução, permitindo fornecer um sistema com elevado desempenho.

Palavras-chave: MixCloud, middleware, repositórios chave/valor, replicação, cloud

Abstract

In the last few years, a variety of key/value storage systems oriented to the *cloud* has been developed. More specifically, these systems are core components of applications for the *cloud*, providing them functional correctness, availability, performance and scalability. Performance resulting from solutions might vary with the type of operations involved. On the other hand, their programming interfaces differ, resulting in an impact on the design of the applications and their commitment to a specific system. In this context, there are systems that does not provide internal replication, with a significant impact on the availability of data and performance. The main objective of this work is to present the new MixCloud middleware system that offers a replicated key/value storage service. To achieve this, the system implements different existing key/value storage services, allowing for a combination of their best features. Our solution offers an interface independent from the systems considered here. The mechanism of replication follows a model of consistency, based on the application of transformation of operations. We accomplish to provide a system with higher performance by returning the result of an operation as soon as the fastest storage system finish its execution.

Keywords: MixCloud, middleware, key/value storage system, replication, cloud

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Introdução

As plataformas de *Cloud* permitem ao conjunto de fornecedores disponibilizar recursos, sob a abstracção de serviços, com custos de implementação e manutenção competitivos. Por seu lado, os utilizadores acedem, via serviços e de forma transparente, a esses recursos pagando uma taxa consoante a sua utilização. Estes recursos podem ser desde aplicações a servidores de suporte ou até mesmo canais de comunicação, encapsulados em serviços para abstracção do utilizador da sua localização e organização.

Neste domínio, têm sido desenvolvidos um grande número de repositórios chave/-valor. Este tipo de repositórios difere significativamente das bases de dados convencionais, não adoptando o seu modelo de consistência total nem o modelo relacional. Tal acontece pois um modelo de dados simples, baseado em chave-valor, facilita a sua distribuição e replicação, associado a uma maior disponibilidade dos dados, com um trade-off da consistência dos mesmos. Outra vantagem deste modelo é uma escalabilidade maior perante o aumento do volume de dados e do número de clientes. As diversas aproximações propostas, como o BigTable [CDG+08], Dynamo [DHJ+07], Cassandra [LM10] e PNUTS [CRS+08] apresentam diferenças ao nível do desenho e da arquitectura, assim como do próprio modelo de dados e operações disponibilizadas, as quais se adaptam aos objectivos específicos. Estes serviços oferecem assim garantias de fiabilidade, disponibilidade, segurança, e desempenho distintos. Estas são ainda influenciadas pelas características do meio de acesso, tipicamente via internet, utilizado pelo cliente destes serviços.

Ao nível de desempenho, alguns sistemas apresentam melhores indicadores na presença de operações de leitura enquanto outros nas de escrita. Tal deve-se ao facto de a taxa de atendimento e a sua latência variar entre sistemas consoante o tipo de operação.

Por outro lado, as interfaces de programação disponibilizadas diferem e reflectem o modelo de dados destes sistemas. Estas diferenças influenciam o desenho das aplicações comprometendo estas a um repositório específico. Torna-se assim difícil a resposta a alterações nos custos dos serviços ou o aproveitamento de novos serviços oferecidos por novos fornecedores.

O uso simultâneo de um conjunto de sistemas que replicam os dados armazenados permite mitigar os problemas antes apontados e melhorar o desempenho geral, na medida em que se torna possível agregar as melhores características que cada um tem para oferecer no que toca à execução de diferentes tipos de operações e colmatar falhas em alguns sistemas pela utilização dos restantes. Adicionalmente, abre a possibilidade de distribuir a carga entre os vários sistemas, implementar algoritmos de consenso para lidar com falhas bizantinas ou introduzir cifra e fragmentação dos dados entre os vários repositórios para aumentar a segurança e privacidade [BCQ+11].

Neste trabalho é apresentado o sistema MixCloud, um sistema *middleware* que implementa um serviço de armazenamento replicado e distribuído, o qual integra diversos repositórios chave-valor, de forma a explorar as vantagens de cada um. Este serviço está vocacionado para utilização de repositórios sem garantias de replicação interna. A utilização de vários repositórios permite implementar mecanismos de replicação para aumentar a disponibilidade do serviço. Para tal, as operações de escrita dos clientes são executadas em todos os repositórios. Um desafio importante do sistema consiste em garantir a convergência final dos dados armazenados nos diversos repositórios, com o mínimo de coordenação dos mesmos. Para esse fim, o sistema utiliza um algoritmo de transformação de operação que permite a execução imediata das operações recebidas e garantir consistência eventual das réplicas.

Mais concretamente, o MixCloud tem por objectivo implementar um serviço de armazenamento de dados que responda aos problemas seguintes:

- Implementar uma solução com suporte de replicação dados sobre repositórios chave/valor sem garantias de replicação interna;
- Um modelo de replicação segundo o modelo "weakly consistent"oferecendo um serviço escalável de alto desempenho e disponibilidade, perante falhas dos fornecedores ou rede e, eventualmente, contra dados corrompidos, falhas bizantinas, etc.
- Uma interface de programação independente da plataforma, combinando os diversos sistemas entre si e oferecendo independência da aplicação relativamente aos fornecedores de serviços.
- Explorar um aumento de desempenho distribuindo a carga pelos repositórios e esperar apenas pela resposta da operação recebida do repositório mais rápido na sua execução.

Assim, na secção 2 desta dissertação são apresentados modelos de replicação, alguns

sistemas de ficheiros distribuídos, exemplos de repositórios na Cloud e uma breve introdução à transformação operacional. A solução adoptada neste projecto é detalhada na secção 3, onde o desenho deste sistema em termos de modelo de dados, interface, a API, a sua arquitectura e funcionalidade são descritas. A transformação operacional orientada para esta solução vai ser definida nesta secção. A implementação do sistema vai ser discutida na secção 4 e os seus resultados avaliados na secção 5. Por fim, apresentam-se ao longo da secção 6 algumas conclusões e o trabalho futuro.

2

Trabalho Relacionado

Neste capítulo introduzimos sistemas de armazenamento e mecanismos que será essencial no nosso projecto. Adicionalmente abordamos outros sistemas, a fim de se perceber as diferentes soluções de cada um tendo em conta os objectivos específicos a que se propõem. Iremos falar do mecanismo de replicação como forma de aumenta a disponibilidade e desempenho de um sistema distribuído, apresentando duas abordagens diferentes. Em seguida, apresentamos brevemente alguns sistemas de armazenamento distribuídos que têm por base no seu desenho o modelo tradicional dos sistemas de ficheiros comuns (i.e. Unix File System/POSIX). Por fim apresentamos exemplos de repositórios pensados para grandes escalas, como a *cloud*, focando as diferenças entre si ao nível do desenho, arquitectura, modelo de dados e API. O leitor que esteja familiar com estes conceitos pode seguir para o capítulo seguinte.

Os sistemas distribuídos apresentam necessidades estritas de desempenho, disponibilidade e tolerância a falhas, de forma a torna-lo robusto perante alterações no ambiente envolvente (i.e falhas nos canais e componentes). Desta forma foi proposto num mecanismo de replicação, que através da distribuição de cópias de um elemento pelos componente, permite atender a essas necessidades. Com o aparecimento da *cloud* houve necessidade de criar sistemas que conseguissem funcionar correctamente numa grande escala e ainda oferecer graus de desempenho e disponibilidade elevados, fazendo um trade-off com a consistência.

2.1 Replicação de dados

A replicação surge como uma solução viável para tornar os sistemas tolerantes a falhas e promover um aumento do desempenho e disponibilidade da informação. A ideia consiste em manter várias cópias de um objecto distribuídas em locais distintos no sistema. Na visão do cliente a replicação é transparente, apenas existe uma cópia do objecto no sistema, permitindo ao sistema distribuir os clientes pelas cópias e mascarar as falhas reencaminhando-os sempre para uma cópia disponível. A utilização desta solução introduz problemas de consistência complexos de resolver.

As soluções existentes para a replicação podem dividir-se em dois tipos: pessimista e optimista. Na solução pessimista as réplicas sincronizam as alterações produzidas num objecto entre si antes de prosseguirem na execução, oferecendo uma consistência forte. Por outro lado, a solução optimista relaxa o grau de consistência e permite a execução de operações em concorrência, sob possibilidade de ocorrerem divergências entre réplicas e estas tornarem-se visíveis ao cliente. Nesta aproximação a actualização do objecto é efectuada apenas na réplica primária e a propagação das alterações é realizada posteriormente em background para as réplicas secundárias. Na detecção de conflitos aplicam-se mecanismos de reconciliação das versões divergentes numa versão final do objecto. A solução optimista distingue-se positivamente da pessimista oferecendo uma maior disponibilidade e desempenho ao sistema subjacente. Para aplicações com exigências estritas de consistência a solução pessimista é a mais indicada.

2.2 Sistemas de ficheiros distribuídos

No domínio dos sistemas distribuídos existem sistemas que exploram soluções para possibilitar o armazenamento e partilha de dados em rede, chamados sistemas de ficheiros distribuídos. Os objectivos a que este tipo de sistemas se propõem focam essencialmente a agregação de recursos computacionais (i.e. servidores) para explorar uma maior capacidade de armazenamento, incentivar a partilha entre utilizadores, permitir uma gestão administrativa centralizada dos dados, oferecer mecanismos de distribuição da carga e tolerância a falhas. O serviço que implementam disponibiliza uma interface de acesso bem definida para que clientes e administradores possam emitir os seus pedidos e realizar funções de gestão dos dados, mantendo o estado coerente e correcto a cada operação aplicada. A organização lógica dos dados segue o modelo tradicional dos sistemas de ficheiros comuns (i.e. Unix File System/POSIX), segundo uma árvore hierárquica de nomes. Cada nó da árvore corresponde a uma directoria ou a um ficheiro. A interface de acesso implementa as principais operações de gestão, manipulação e pesquisa. A arquitectura de base é semelhante para todos os sistemas: existe um cliente que efectua pedidos e um servidor que armazena os dados do cliente e responde aos seus pedidos.

A implementação destes sistemas apresentam diversas problemáticas, sendo necessário introduzir requisitos para garantir as propriedades do sistema. O requisito de transparência surge como um dos mais importantes, consistindo na capacidade de abstrair e esconder dos clientes pormenores de implementação do sistema. Em particular, a aplicação do utilizador não deve necessitar de ser modificada devido a alterações nos componentes ou na distribuição dos dados. Ao nível da transparência de acessos garante-se que a distribuição dos dados não é conhecida. Um outro requisito necessário é a replicação dos dados que, como dito na secção anterior, apresenta uma solução imediata para aumentar a disponibilidade da informação. O requisito de segurança estabelece o nível de acesso aos dados, negando o acesso a utilizadores aparentemente faltosos ou não autorizados. Por último, um sistema deve também apresentar eficiência e desempenho comparáveis com os sistemas de ficheiros convencionais.

Nos sistemas apresentados nesta secção assume-se um modelo assíncrono em que o sistema é constituído por um conjunto de réplicas que trocam mensagens entre si. Assume-se o modelo de falhas *fail-stop* e que não existe memória partilhada ou relógios sincronizados.

2.2.1 Coda

O *Coda* é um sistema de ficheiros distribuído que segue a visão tradicional dos sistemas do mesmo domínio, implementando um repositório de dados remoto para um ambiente distribuído de acesso transparente na visão da aplicação. O seu desenho foca essencialmente a escalabilidade, alta-disponibilidade dos dados, desempenho e elevado grau de consistência. Providência resiliência a falhas implementando uma estrutura robusta e mecanismos de replicação, distribuição de carga e tratamento de desconexão. O modelo de falhas adoptado está assente sobre uma estratégia optimista onde assume-se que existem poucos ou nenhuns conflitos, podendo estas ser corrigidos caso ocorram.

Arquitectura

O sistema *Coda* subdivide-se em dois componentes para implementar o serviço de armazenamento de dados: *Venus* e *Vice*, apresentado na figura 2.1. O componente *Venus* funciona como *front-end* para o cliente e gere as suas réplicas armazenadas na cache local. Desempenha o papel de *front-end* pois abstrai o cliente da implementação do sistema e gere a cache local de forma a manter as réplicas coerentes com o estado armazenado nos servidores. Por sua vez, o *Vice* executa no *back-end* e é responsável por implementar o serviço de acesso aos objectos e fazer a sua gestão.

Os objectos lógicos (ficheiros) são replicados fisicamente por um grupo de servidores, *Volume Storage Group* (*VSG*), de onde apenas um sub-grupo se encontra activo, *Active VSG* (*AVSG*), que atende os pedidos dos clientes a cada momento. O *AVSG* altera-se consoante os servidores vão ficando inacessíveis devido a falhas na rede ou falhas internas. No lado do cliente é utilizada uma cache que armazena cópias parciais ou integrais de

ficheiros obtidos a partir dos servidores. Este mecanismo permite diminuir o número de acessos ao sistema remoto, sendo as operações de leitura e escrita aplicadas directamente na cópia da cache, caso exista. Adicionalmente permite suportar operações desconectadas. As operações executadas concorrentemente são propagadas entre clientes utilizando um mecanismo de *callback*, um cliente regista no servidor a intenção de ser notificado perante alterações no ficheiro.

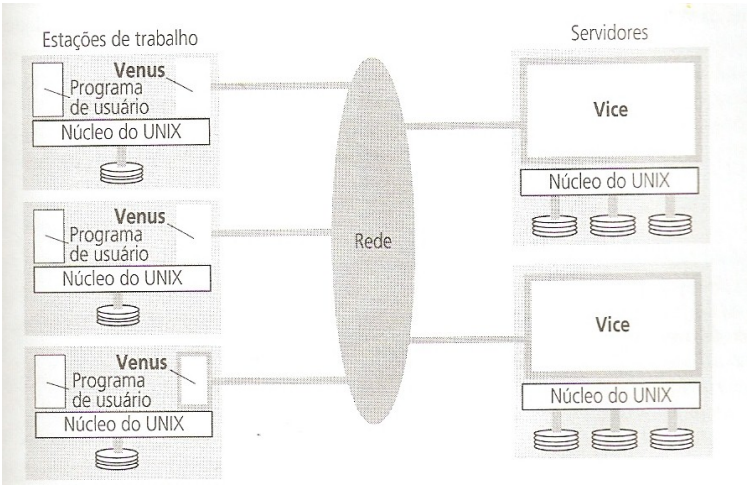

Figura 2.1: Organização dos componentes Venus/Vice no *Coda* (de [CDK07])

Assume-se uma aproximação optimista no modelo de replicação permitindo a um cliente efectuar alterações mesmo na presença de falhas no sistema ou em períodos de desconexão. A cada réplica é associado um vector versão (*CVV*) que representa o seu histórico de actualizações, armazenando uma estimativa do número de modificações efectuadas em cada servidor do *SVG*. Diz-se que não existe conflicto entre dois *CVV* quando para todas as posições de um deles, v_i, e todas as posições correspondentes, v_j, no outro, $v_i \geq v_j$. Caso tal não se verifique, foi detectado um conflito e cada uma das réplicas não reflecte pelo menos uma actualização realizada na outra. A réplica em conflito é marcada como "inoperante"e o proprietário do ficheiro é contactado para efectuar a resolução manualmente.

O modelo de acessos segue uma variante da estratégia *read-one/write-all*. Perante uma operação de abertura do ficheiro o cliente verifica inicialmente se possui uma cópia na cache local. Apenas se não existir, escolhe um servidor preferido entre os *AVSG* do ficheiro, e solicita uma cópia dos atributos e conteúdos do ficheiro. Comunica em seguida com os restantes membros do *AVSG* para verificar se a cópia recebida corresponde à versão mais recente disponível. Se não corresponder, selecciona um novo servidor preferido com a versão da réplica mais recente e o conteúdo é transferido novamente. Uma notificação é envida aos membros do *AVSG* se existirem servidores com réplicas desactualizadas.

Após o fecho de um ficheiro, o cliente envia os conteúdos modificados e o *CVV* corrente em paralelo para todos os membros do *AVSG*. Este passo permite maximizar a probabilidade de todos os servidores manterem uma versão actualizada da réplica o tempo todo. Se o *CVV* recebido for superior ao mantido actualmente, o novo conteúdo é armazenado e sinaliza o cliente com o término da operação. No entanto, se um conflito de versões existir, o procedimento a seguir é igual ao descrito acima. A execução segue no caso positivo, o servidor comunica as alterações a todos os clientes dos quais possua um registo de *callback*. O cliente que emitiu o fecho do ficheiro calcula um novo *CVV* com base nas respostas dos servidores, incrementando em unidade a sua posição se responderam positivamente à mensagem de actualização, e distribui pelos membros do *AVSG*.

2.2.2 Hadoop Distributed File System

O Hadoop Distributed File System (*HDFS*) [Had11] é um sistema de armazenamento de ficheiros pensado para grande volumes. Implementa um serviço de armazenamento de elevado desempenho e largura de banda nos acessos indicado para aplicações com padrões de acesso a múltiplos ficheiros em simultâneo. O mesmo pode ser comprovado pela sua adopção na *framework* de trabalho *map-reduce* Hadoop que promove um alto grau de eficiência quando utilizado para pesquisar sobre um grande volume de dados distribuído pelo sistema. Este modelo segue a estratégia de divisão do trabalho pelos repositórios e no fim agrega as diferentes respostas de forma a paralelizar e maximizar o processo de pesquisa. O HDFS disponibiliza também uma API de acesso que contém primitivas para manipulação directa dos dados e gestão do sistema de ficheiros. Mesmo na presença de potenciais falhas dos componentes os acessos são garantidos, apresentando uma estrutura robusta e implementando mecanismos de replicação que mantêm o sistema disponível. A organização lógica dos dados no sistema segue o modelo do sistema de ficheiros UNIX. O desenho do HDFS baseia-se no sistema Google File System (GFS) [GGL03].

2.2.2.1 Arquitectura

A arquitectura do sistema *HDFS* apresenta uma estrutura organizada sob o modelo *master/slave*, como se verifica na figura 2.2. O componente *master* corresponde ao nó *NameNode* responsável por gerir o espaço de nomes e distribuição dos ficheiros. O conjunto dos *slaves* é formado por *DataNodes* que armazenam persistentemente os ficheiros e respondem aos pedidos dos clientes. Cada ficheiro é internamente representado por conjuntos de blocos que guardam parcialmente o conteúdo do ficheiro original. Os *DataNodes* organizam-se em grupos chamados *racks*, tendo em conta a proximidade geográfica, pelos quais são distribuídos os blocos com o intuito de obter melhor distribuição da carga. Os blocos são replicados pelos diversos *racks* de forma a garantir disponibilidade e tolerância a falhas do serviço.

Como representado na figura 2.2, o cliente contacta inicialmente o *NameNode* para informar qual o tipo da operação que pretende efectuar (leitura ou escrita). Na presença de uma leitura, o *NameNode* responde ao cliente com a localização dos *DataNodes* que armazenam os blocos dos ficheiros a aceder. Na presença de uma aplicação *map-reduce*, o cliente transfere para cada um o trabalho a executar. Numa primeira fase cada *DataNode* executa em paralelo a fase *map* que consiste em pesquisar e retornar o conteúdo de cada bloco que respeita as restrições. Em seguida os resultados parciais são agregados e colocados num ficheiro criado no *HDFS* para esse fim, sendo esta a fase de *reduce*.

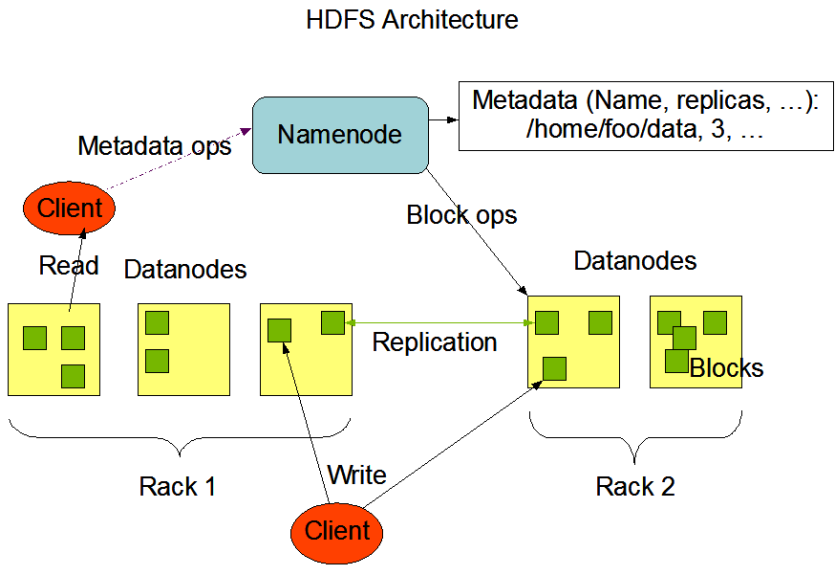

Figura 2.2: Organização dos componentes no *HDFS* (de [Had11])

O cliente só inicia o pedido de escrita de um ficheiro quando o primeiro bloco de informação se encontra completamente preenchido. Por sua vez o *NameNode* regista o nome do ficheiro no espaço de nomes, caso não exista, e indica qual o *DataNode* onde será armazenado o bloco. O processo repete-se para os restantes blocos. Após transferir todos os blocos, cliente contacta de novo o *NameNode* que procede ao *commit* da operação. Um ficheiro após fechado não pode ser mais alterado, no entanto as leituras não possuem limite de ocorrências. A adopção deste modelo de consistência permite simplificar a manutenção da coerência dos dados e aumentar débito do sistema.

A responsabilidade da gestão de sessões está a cargo do *NameNode* que mantém uma visão dos nós activos no sistema a cada momento. Utiliza-se um mecanismo de *heartbeats* periódicos, onde cada *DataNode* informa o *NameNode* da sua disponibilidade para receber pedidos e quais o blocos que armazena. Em caso de falha na recepção são invocadas operações para replicação dos blocos que o *DataNode* mantinha, com o intuito de manter o sistema estável e coerente.

2.2.3 Discussão

No domínio dos sistemas de ficheiros distribuídos são propostas diversas aproximações com o objectivo comum de oferecer um serviço com capacidade de armazenamento superior e elevada disponibilidade dos dados, com suporte para mecanismos de distribuição de carga e tolerância a falhas. Estes serviços oferecem uma interface de acesso bem definida que mantém o estado coerente e correcto a cada operação aplicada. O modelo de dados utilizado segue o modelo tradicional de sistemas de ficheiros comuns, com o espaço de nome hierarquizado.

Nas subsecções anteriores foram apresentados dois sistemas distintos: *Coda*, adopta uma estrutura descentralizada onde os servidores são agrupados segundo a distribuição dos ficheiros; *HDFS*, implementa o modelo master/slave, onde um servidor principal (*master*) gere o espaço de nomes e os acessos a dados e um grupo de servidores (*slaves*) armazenam os ficheiros. Em ambos os sistemas os dados são replicados pelo grupo de servidores para garantias de disponibilidade, distribuição da carga e maior eficiência no tratamento de pedidos, tornando o sistemas escaláveis com o aumento do volume de dados.

A estrutura do *Coda* aparenta ter um maior factor de crescimento relativamente ao *HDFS*, pois neste ultimo o componente *master* apresenta um ponto único de falha e engarrafamento com o aumento do número de clientes. Por outro lado, o *Coda* apresenta uma mecanismo ineficiente de propagação das alterações nos conjuntos de servidores, na medida em que os clientes só tomam conhecimento dessas alterações após questionarem periodicamente os servidores activos. Ao existirem alterações o cliente deve marcar com inválidas todas as réplicas na cache conhecidas por esse conjunto. Este processo afecta directamente o desempenho geral do *Coda*. A utilização de um servidor central no *HDFS* permite optimizar o registo de novos servidores e a distribuição da carga.

O *Coda* implementa um modelo de replicação optimista enquanto o *HDFS* segue pela vertente pessimista. No primeiro os ficheiros são unidades indivisíveis armazenadas inicialmente em apenas um servidor, que posteriormente, em *background*, propaga as alterações para os restantes. No segundo os blocos que formam um ficheiro são replicados um a um em paralelo pelos demais servidores. Tornar-se visível que o *Coda* apresenta uma maior eficiência no tratamento de operações de escrita abstraindo o cliente do processo de replicação. Relativamente às operações de leitura, o *HDFS* apresenta o modelo *map-reduce* que permite a execução de tarefas em paralelo por conteúdo; mas, o *Coda*, impõe a participação de todos os servidores para garantir que a resposta está correcta. Conclui-se que o *Coda* está orientado para aplicações com elevada percentagem de escritas nos ficheiros enquanto o *HDFS* serve melhor aplicações que necessitem elevado desempenho nas leituras para grandes volumes de dados e processo paralelo independente sobre os dados. Na tabela 2.1 apresenta-se um resumo do que foi discutido nesta secção.

	Coda	HDFS
Replicação	Sim	Sim
Arquitectura	Descentralizada (*AVSG*)	Centralizada (*NameNode*)
Suporta Updates	Sim	Não
Suporta Modelo Map-Reduce	Não	Sim
Suporta Desconexão	Sim	Não

Tabela 2.1: Resumo sobre os sistemas distribuídos

2.3 Repositórios na *cloud*

As plataformas de *cloud* permitem ao conjunto de fornecedores disponibilizarem recursos, sob a abstracção de serviços, com custos de implementação e manutenção competitivos. Neste domínio, os sistemas de armazenamento são fundamentais e têm-se revelado os de maior sucesso e divulgação, oferecendo as mais varias soluções para armazenamento persistentemente da informação, quer dos utilizadores, quer das aplicações [AFG+09]. Nos últimos anos têm sido desenvolvidos um grande número de repositórios chave/valor para suportar os serviços de *cloud*.

Este tipo de repositórios difere significativamente das bases de dados convencionais, não adoptando o seu modelo de consistência total nem o modelo relacional. Tal acontece pois um modelo de dados simples, baseado em chave-valor, facilita a sua distribuição e replicação, permitindo igualmente uma melhoria de desempenho. O sistemas que apresentaremos nesta secção apresentam diferenças ao nível do desenho e arquitectura, assim como o modelo de dados e operações disponibilizadas, as quais se adaptam aos objectivos específicos para os quais foram desenvolvidos. Estes serviços oferecem assim garantias de fiabilidade, disponibilidade, segurança e desempenho, distintas. Estas são ainda influenciadas pelas características do meio de acesso, tipicamente via internet, utilizado pelo cliente destes serviços.

2.3.1 BigTable

O sistema *BigTable* [CDG+08] consiste num repositório distribuído orientado para a *cloud* pensado para suportar grandes volumes de dados estruturados. Entre os seus objectivos procura oferecer alta aplicabilidade, escalabilidade, disponibilidade e desempenho aceitáveis. Apresenta uma estrutura flexível de suporte a diferentes tipos de aplicação, com exigências desde o tamanho de dados a armazenar até a baixas latências de resposta para sistemas de tempo real. Ao contrário das bases de dados convencionais, o *BigTable* não suporta o modelo relacional adoptando um modelo dinâmico, mais simples, que permite aos clientes ter um maior controlo sobre a organização dos seus dados. Para tal implementa uma interface simplificada baseada em acessos chave/valor.

Modelo de Sistema

O modelo de dados adoptado pode ser visto como uma tabela ordenada e esparsa formada por conjuntos de três componentes: coluna, família e linha. A coluna funciona como a unidade de acesso básica que armazena valores indexados por uma chave. Conjuntos de colunas são associadas segundo um domínio comum a que chamamos família, identificada por uma chave única no sistema. As chaves das colunas seguem a sintaxe *família:coluna*. Ao nível das famílias é feito o controlo de acessos das aplicações. Cada linha está associada a um conjuntos de colunas que representam os seus atributos, semelhante às tabelas relacionais, podendo esses atributos serem transversais a várias famílias. As linhas são indexadas por chaves formadas por uma palavra de tamanho variável e agrupadas em conjuntos de *tablets*. O modelo não impõe um relação rígida entre componentes, assim cada linha pode apresentar conjuntos de colunas distinto das restantes. O *BigTable* utiliza estampilhas associada a cada célula da tabela para suportar múltiplas versões de dados. Apenas existe uma tabela no sistema especificada estaticamente antes do sistema iniciar.

Arquitectura

A arquitectura do *BigTable* é constituída por três componentes essenciais: serviço *Chubby*, o *master server* e vários *tablet server*. O serviço *Chubby* consiste num serviço de *locking* que armazena e gere os registos de sessões de trabalho. A gestão administrativa do sistema está a cargo do *master server*, que entre outras funções, distribui a carga pelo sistema e detecta falhas nos componentes. Existem diversos *tablet servers* responsáveis por armazenar um conjunto de *tablets* e atender os pedidos que lhes são dirigidos. A organização hierárquica destes componentes permite distribuir a carga e optimizar a eficiência das pesquisas. Num primeiro nível encontra-se apenas o *root* que mapeia a localização dos *tablets* que formam a tabela de meta-dados do nível seguinte. A tabela de meta-dados mantém um índice actualizado das *tablets* dos utilizadores localizados no ultimo nível. Cada *tablet* é atribuída apenas a um servidor o que permite manter um nível de consistência forte. Toda a informação é armazenada utilizando o sistema de ficheiros Google File System [GGL03], que internamente replica os dados a fim de suportar a recuperação dos mesmo em caso de falha de um *tablet*.

Num contexto de funcionamento normal, o cliente contacta inicialmente o serviço *chubby* para criar uma nova sessão de trabalho que retornar a localização do *root*. A cada novo pedido o cliente contacta o *root* que através de uma pesquisa pela hierarquia de índices obtém a localização do servidor final responsável pelo *tablet* a aceder. Esta informação é armazenada na cache do cliente para no futuro poder aceder directamente ao servidor final. O cliente envia directamente a operação para o servidor. Se for uma operação de escrita, o servidor regista as alterações no seu log sequencial em disco e em seguida na tabela de dados ordenada *memtable* representada em memória. Quando a

memtable atinge um tamanho pré-definido é transformada num ficheiro *SSTable* e armazenada persistentemente no repositório subjacente. Na presença de uma leitura efectua-se a pesquisa numa visão conjunta da *memtable* e *SSTables* em disco e a resposta é retornada ao cliente. Em ambas as operações, antes de executar, procede-se a uma validação prévia da operação. Os acessos a um *tablet* dentro de uma sessão são sempre atómicos.

O componente *master server* é responsável por atribuir *tablets* aos *tablet servers*, detectar adições ou falhas de *tablet servers* e balancear a carga do sistema. Em caso de falha de um *tablet server*, o *master* comunica com o *chubby* para revogar a sessão do servidor e contacta o *GFS* para obter os *tablets* que lhe estavam afectados e distribui pelos restantes. Desta forma consegue-se garantir disponibilidade dos dados e adicionar um grau de transparência a falhas necessário a uma visão coerente e correcta do sistema. Um bom rácio de distribuição dos *tablets* pelos servidores é importante para conseguir alcançar um melhor desempenho e escalabilidade.

2.3.2 HBase

No seguimento do projecto *BigTable* (em 2.3.1) surge o sistema *HBase*, uma nova base de dados distribuída escalável a grandes volumes de dados com suporte a aplicações de tempo real. Tal como o seu antecessor, implementa a noção de distribuição da carga e replicação mantendo os dados sempre disponíveis mesmo perante situações de falha dos componentes. Utiliza um novo repositório de dados, o *Hadoop File System* (ou simplesmente *HDFS*) [Had11].

O modelo de dados adoptado assemelha-se ao *Bigtable*: os dados são organizados por um conjunto de tabelas e indexados por uma chave na forma *família:linha:coluna* que indica a sua localização. As famílias são formadas por conjuntos de colunas e as linhas estão associadas a conjuntos de atributos, correspondentes às colunas. Este modelo apresenta uma estrutura dinâmica onde as linhas podem ter atributos distintos e organizam-se em *tablets* como no *BigTable*. O *Hbase* suporta o armazenamento multi-versões dos dados que são diferenciadas por uma estampilha associada a cada versão.

Arquitectura

A arquitectura do HBase é formada por dois componentes principais: *HBaseMaster* e *HRegionServers*. O componente central *HBaseMaster* corresponde ao componente *master server* e o conjunto dos *HRegionServer* aos *tablet servers* do sistema *BigTable*. Em ambas as situações encontram-se organizados da mesma forma desempenhando funções semelhantes.

Adicionalmente são utilizados dois sistemas complementares: o *HDFS* e *Zookeepers* [Zoo11]. O sistema *HDFS* e *Zookeeper* desempenham os mesmos papeis, respectivamente, que o *Google File System* e o serviço *chubby* do *BigTable*.

O cliente quando emite um novo pedido (leitura ou escrita) contacta inicialmente o *Zookeeper* para obter a localização do *HRegionServer* que representa o *root*; em seguida questiona o *root* e percorre a árvore de indexação de forma a obter o endereço do servidor

responsável pelo *tablet* a aceder. Esta informação é armazenada em cache no cliente para futuros acessos directos ao servidor final. O cliente contacta o servidor e caso se trate de uma operação de escrita, a alteração é registada num log sequencial em disco e em seguida numa estrutura em memória que representa a tabela, denominada *MemStore*. Se a *MemStore* alcançar um determinado limite de tamanho, o conteúdo é copiado para um ficheiro e armazenado no *HDFS*. Caso se trate de uma operação de leitura, a pesquisa é efectuada sobre uma visão consistente formada pela *MemStore* e os dados em ficheiro. O processo é eficiente pois os *tablets* encontram-se ordenados por chave em ambas em estruturas e utilizam-se índices para indicar a localização das chaves.

A gestão é centralizada e efectuada pelo componente *HBaseMaster*. Realiza a gestão das sessões dos *HRegionServers* e distribui a carga no momento do seu registo. No caso de um dos servidor falhar inicia o processo de distribuição das *tablets* pelos restantes e actualiza os seus meta-dados. Este mecanismo de distribuição permite aumentar a disponibilidade da informação e melhorar o desempenho do sistema.

2.3.3 Dynamo

O *Dynamo* [DHJ+07] surge como um repositório de dados de alta disponibilidade para armazenamento de dados chave/valor. Oferece uma plataforma robusta e eficiente, escalável a grandes volumes, cujo o acesso é eficiente e efectuado com baixas latências. As propriedades de disponibilidade e escalabilidade são alcançados pelo uso de técnicas de particionamento e replicação dos dados. O seu desenho foi pensado de forma a conseguir garantir fiabilidade e disponibilidade dos dados em termos de uma solução baseada em consistência eventual, mantendo o sistema sempre operacional e receptivo a operações mesmo na presença de falhas. Este sistema é usado para suportar diferentes tipos de serviços que necessitam de um mecanismo de controlo apertado para estabelecer o trade-off entre disponibilidade, consistência e desempenho. Apresenta uma arquitectura descentralizada ao nível da gestão sem recurso a um ponto central.

Modelo de Sistema

Os dados introduzidos pelo utilizador são identificados por uma chave única e um elemento que representa a versão e distribuídos através dessa chave pelo sistema. A chave e o valor são representados por arrays de bytes opacos. A interface de acesso aos dados apenas expõe duas primitivas: *get* e *put*. A operação *get* permite a partir de uma chave localizar e retornar o valor associado e a operação *put* permite armazenar um valor dado em argumento juntamente com a chave que o identifica.

O *Dynamo* implementa um modelo de consistência eventual que permite a propagação assíncrona de escritas por todas as réplicas. Nestes domínios podem ocorrer falhas nos componentes ou nos canais de comunicação ou apenas tempos diferentes de propagação, que leve a uma deficiente propagação das alterações entre réplicas e torne visível

estados inconsistentes nas leituras. O sistema utiliza um mecanismo de versões suportado por relógios vectoriais [Lam78] para capturar a causalidade do eventos. Quando uma nova chave é adicionada ou alterada é criada uma nova versão. Um novo conflito surge quando durante uma leitura são detectadas cópias de uma mesma chave que apresentam alterações e não se reflectem nas restantes, logo conteúdos diferentes. Tal pode ser visto através da comparação dos relógios vectoriais associados a cada cópia. No contexto de funcionamento normal, as novas versões substituem as antigas (reconciliação sintáctica). Na presença de conflitos é pedido ao utilizador que cria uma versão estável a partir das cópias que lhe são devolvidas pela leitura (reconciliação semântica).

Arquitectura

O modelo de arquitectura adoptado pelo *Dynamo* segue a filosofia do sistemas *peer-to-peer* (P2P): descentralizada, estruturada e orientada ao serviço onde todos os nós assumem a mesma responsabilidade. Implementa uma *distributed hash table* (DHT) onde o espaço de nomes é circular, ordenado e particionado pelos diferentes nós. Os identificadores (chaves) dos nós são calculados através de uma função de consistent hashing [KLL+97] que indica a sua posição no circulo.

Todos os nós são responsáveis por uma partição de chaves, limitada pelo identificador anterior e o próprio. A sua função é responder aos pedidos dirigidos às chaves da sua partição. Em caso de falha deste, um outro nó assume a sua responsabilidade temporariamente e armazena as alterações efectuadas. O nó principal ao regressar é actualizado pelo nó auxiliar. Na figura 2.3 apresenta-se uma possível distribuição de identificadores pelos nós de A a G.

As partições são replicada por um número arbitrário N de nós do sistema de forma garantir disponibilidade dos dados em case de falha do nó principal. A actualização das réplicas é feita da seguinte forma: ao receber um novo pedido, o nó principal armazena uma cópia localmente e propaga cópias para os $N-1$ nós sucessores no anel. Cada nó fica responsável pela região acumulada dos $N-1$ nós antecessores e a própria. Um exemplo de replicação de chaves é demonstrado na figura 2.3 com a chave K a ser replicada pelas partições B, C e D, sendo $N=4$.

Neste sistema é permitido aos clientes enviar pedidos para qualquer nó do sistema. Se o nó receptor não for responsável pela partição a aceder, re-encaminha o pedido para o destino final em apenas um passo. Um coordenador da partição ao receber uma operação de escrita, gera um vector versão e armazena a nova versão localmente. Em seguida replica a versão para os restantes coordenadores e espera que se forme um quórum com pelo menos W respostas à escrita, sendo W um número mínimo de participantes na escrita. Se não houver respostas suficientes a operação é anulada. Na presença de uma leitura o coordenador pede aos restantes a versão actual dos dados a aceder. Espera no mínimo por R respostas, sendo R o número mínimo de participantes, e apenas envia as versão distintas ao cliente. Se houver múltiplas versões o cliente efectua a reconciliação

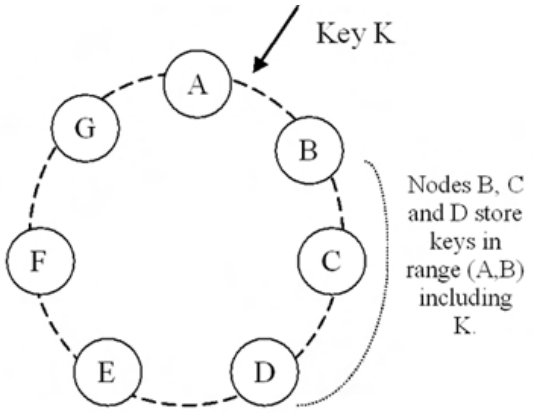

Figura 2.3: Partição e Replicação das chaves no anel (de [DHJ+07])

e envia um pedido de escrita com a versão final.

As sessões de trabalho são geridas por um serviço de *gossip* que propaga e mantém uma visão eventualmente consistente dos nós pertencentes ao sistema. Cada nó mantém persistentemente uma lista de nós activos que conhece a cada momento e propaga periodicamente essa informação através do *gossip*. Quando um nó pretende juntar-se ao sistema, emite um pedido de registo a um nó activo que por sua vez procede ao re-arranjo das partições e propaga as alterações.

2.3.4 Cassandra

No domínio dos sistemas distribuídos de armazenamento surge o sistema Cassandra [LM10], pensado para implementar um base de dados distribuída de armazenamento de grandes volumes de dados em larga escala (distribuída por vários centros de dados). Oferece um serviço eficiente de baixa latência que explora alto *throughput* nas escritas, sem sacrificar a eficiência das leituras, e providencia alta disponibilidade sem pontos único de falha. O estado do sistema é mantido persistente e gerido face a eventuais ocorrências de falhas, na rede ou componentes de base, permitindo melhor fiabilidade dos dados e escalabilidade. Baseia-se no sistema *BigTable* (2.3.1) de onde vai buscar o modelo de dados e no sistema *Dynamo* (2.3.3) de onde adopta o desenho.

Modelo de Sistema

O modelo de dados apresentado no *Cassandra* assemelha-se a uma tabela relacional formada por linhas e conjuntos de colunas (atributos) associados a essas linhas. Adicionalmente implementa a noção de família que agrega colunas relacionadas entre si. A inovação aparece pela introdução de um novo tipo de coluna chamado super. As super colunas são indexadas por uma chave, tal como as colunas, e o seu valor corresponde

a um conjunto de colunas simples. Proporciona um novo nível de estruturação dos dados. O acesso a uma coluna simples no corpo de uma super coluna segue a convenção *família:super_coluna:coluna*. As operações de leitura e escrita são atómica sobre uma linha.

Arquitectura

O sistema *Cassandra* apresenta uma arquitectura organizada segundo a filosofia de uma rede P2P estruturada. Implementa no seu topo uma DHT distribuída que distribui os nós por partições do espaço de nomes, podendo uma partição ser replicada por vários nós. A cada nó é atribuído um identificador. Os nós assumem as mesma responsabilidade que os nós no sistema *Dynamo*.

As operações de escrita ao chegarem ao nó final, são registadas no log sequencial em disco e aplicadas as alterações numa estrutura em memória que representa a tabela de dados. Se for uma leitura, é efectuada uma consulta na estrutura em memória acedendo ao disco se não obtiver nenhuma informação. Um índice de chaves é mantido memória para diminuir os tempos de pesquisa em disco. O coordenador de uma partição propaga as operações pelos restantes e espera por S respostas, número mínimo de réplicas que participam na escrita ou leitura, consoante o nível de consistência escolhido. Responde em seguida ao cliente com o resultado final da execução da operação.

Tal como no sistema *Dynamo*, cada nó armazena em disco a sessão de trabalho actual conhecida do sistema. Este modelo permite o encaminhamento eficiente de pedidos por caminhos directos entre nós. Utiliza um serviço semelhante ao *gossip* para propagar alterações no conjunto de nós activos. Implementa o conceito de *hinted writes* em que um nó auxiliar pode assumir o papel temporário de um coordenador em caso de falha deste. Ao retornar, o estado do coordenador é actualizado pelo nó auxiliar.

2.3.5 MongoDB

Uma nova solução que une as características dos sistemas de bases de dados tradicionais e baseados em indexação chave/valor é apresentada no sistema *MongoDB* [Mon11]. Do primeiro domínio adapta o modelo de queries e oferece algumas funcionalidades base e do segundo obtém a rapidez na indexação dos dados e elevada escalabilidade. Apresenta igualmente elevado grau de disponibilidade, robustez com suporte a tolerância a falhas por omissão e flexibilidade de adaptação a vários tipos de serviços. A implementação de um modelo de dados não relacional facilitou o desenho do sistema em termos da replicação, tolerância a falhas e escalabilidade. O seu desenho diverge dos restantes sistemas anteriormente apresentados, optando por um novo modelo de dados, de replicação e distribuição dos componentes.

Modelo de Sistema

Neste sistema foi adoptado um novo modelo de dados orientado ao documento. Um documento é a unidade básica de armazenamento dos dados, sendo identificado por uma

chave única no sistema, organizada lexicograficamente, e o seu conteúdo formado por um conjunto arbitrário de atributos/campos associados a valores. Os valores podem ser de tipos básicos de dados (e.g. inteiros, booleanos e strings), vectores de valores ou então apontadores para outros documentos. Um grupo de documentos forma uma colecção, também identificada por uma chave única. Podem ser vistas como sendo equivalentes às tabelas nas bases de dados tradicionais ou famílias dos repositórios chave/valor. Dentro de uma colecção não existe um esquema estrutural rígido para os documentos, sendo que onde cada um deles pode apresentar diferentes atributos. As colecções são dinamicamente particionadas pelo sistema em grupos de documentos de forma a distribuir a carga.

A interface de acesso oferecida pelo sistema é constituída por várias operações que permitem ler e alterar os dados inseridos na base de dados. Estas operações são poderosas e genéricas permitindo pesquisar documentos de forma eficiente por meio de expressões regulares formadas por documentos que indicam o conteúdo da pesquisa e restrições sobre esse conteúdo. Por exemplo, a operação $find(\{x : 3, y : "foo"\})$ devolve todos os documentos que contenham um campo x com o valor 3 e y com a palavra "foo"enquanto a operação $insert(\{colors : ["blue","black"]\})$ insere um novo documento apenas com um campo "colors" associado aos atributos "blue"e "black". Caso não seja especificado, o identificador de um novo documento é gerado pelo sistema.

Pela primeira vez é apresentado um mecanismo de índices controlado pelo utilizador, permitindo criar novos índices sobre qualquer atributo sem ser a chave primária dos documentos. Estes podem ser usados para optimizar as pesquisas. A ideia é semelhante ao mecanismo de índices encontrado nas bases de dados.

Arquitectura

Na figura 2.4 está representada a arquitectura do sistema *MongoDB*, que se divide três componentes principais: *mongos*, *config servers* e *shard*. Os *mongos* funcionam como ponto de entrada no sistema para os clientes, abstraindo-os da arquitectura interna. Comunicam com o sistema em nome dos clientes re-encaminhando os pedidos até ao servidor final. Um conjunto de *config servers* gere uma visão coerente do estado do sistema, permitindo por exemplo saber a localização de um *shard*. Os servidores finais *mongod* são organizados em grupos de *shards* que armazenam uma partição dos dados do sistema. No interior do *shard* é feita replicação pelos servidores de forma a assegurar a disponbilidade dos dados e recuperação na presença de falhas.

O modelo de replicação implementado segue uma aproximação *master/slave* onde a cada momento apenas um *mongod* funciona como *master* e os restantes como *slaves*. Todas as operações de escrita enviadas para uma partição são recebidas apenas pelo *master*. As alterações produzidas são armazenas persistentemente no *master* e propagadas em background para os restantes servidores. Este modelo oferece garantias de consistência forte pois o *master* executa todas as operações de escrita por uma ordem definida, mantendo

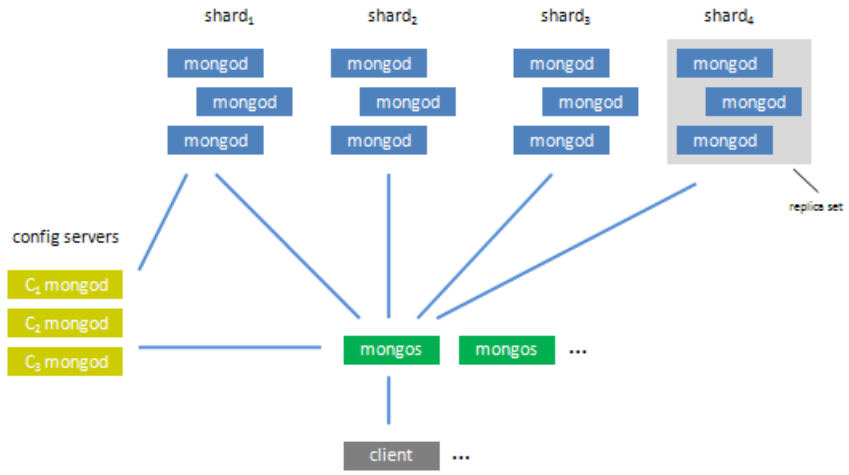

Figura 2.4: Arquitectura do sistema MongoDB (de [Mon11])

sempre um estado correcto. As leituras podem ser atendidas por qualquer servidor da partição desde que o grau de consistência escolhido para a operação não seja forte. Caso contrário terá que ser atendida pelo *master*. No caso de o *master* ficar indisponível, os *slaves* comunicam entre si e seleccionam o novo *master* de entre eles.

Quando um novo *mongos* é adicionado ao sistema contacta inicialmente um *config server* para obter o estado do sistema, armazenado apenas durante o período em que está activo. Todas as alterações que ocorram na configuração serão posteriormente propagadas pelos *config servers* para os *mongos* e os *mongod*.

A configuração arquitectural apresentada permite de uma forma eficiente distribuir a carga e explorar paralelismo na execução das operações para aumento do desempenho do sistema, na medida em que as operações são distribuídas pelos *shards*, consoante a partição a aceder, e os *masters* suportam concorrência no atendimento dos pedidos. Adicionalmente permite de forma simples e directa reforçar a capacidade do sistema adicionando novos servidores sem impacto no desempenho.

2.3.6 Discussão

O estudo realizado em [CDG+08, DHJ+07, CRS+08] revelou que as tradicionais bases de dados relacionais são uma solução longe do ideal para algumas situações, devido à excessiva complexidade, custos associados ao modelo relacional e limitação de disponibilidade devido ao modelo de consistência. Para responder às necessidades de desempenho das aplicações na *cloud* tornou-se necessário criar um novo tipo de repositórios com um modelo de dados simplificado (p.e. chave-valor) e consistência relaxada. Este tipo de sistemas oferecem mecanismos eficientes de distribuição, replicação e recuperação de falhas que permitem garantir fiabilidade, disponibilidade, segurança e desempenho.

Nesta secção foram apresentados vários sistemas que diferem entre si ao nível do

	BigTable/HBase	Dynamo/Cassandra	MongoDB
Modelo de Dados	Chave-Valor	Chave-Valor	Documento
Chave de Acesso	Composta	Simples/Composta	Composta
Modelo de Consistência	Estrito	Relaxado	Estrito
Modelo Org. dos Dados	Arvore B+	DHT	Shards
Modelo de Acesso	Centralizado	Qualquer réplica	Qualquer réplica

Tabela 2.2: Resumo sobre os repositórios na *cloud*

desenho e arquitectura, assim como no próprio modelo de dados e operações disponibilizadas, tendo em conta os objectivos a que se propõem. O *BigTable* e *HBase* propõem uma estrutura hierárquica, em árvore, de servidores pelos quais estão particionados os dados; o *Dynamo* e *Cassandra* implementam uma *DHT* distribuída com espaço de nomes circular pelo qual os servidores são distribuídos; o *MongoDB* utiliza *shards* pelos quais são particionados os dados, formados por múltiplos servidores que replicam os dados internamente. Todos os sistemas com excepção ao *MongoDB* implementam um modelo de dados orientado à chave-valor, podendo a chave ser simples ou composta. Este modelo pode ser adaptado ao *MongoDB* sendo a chave composta por *colecção:documento:atributo*. Um modelo de consistência forte, garantindo a correcção dos dados a todo o momento, foi adoptado por *BigTable* e *MongoDB* em detrimento de alta disponibilidade. Por outro lado os restantes relaxam a necessidade de consistência focando a disponibilidade mesmo perante falhas do sistema, podendo porém, apresentar valores inconsistentes às aplicações.

Os sistemas tipo *Dynamo* possibilitam um maior grau escalabilidade, pois apresentam uma estrutura totalmente distribuída associada a modelos eficientes de adição de novos servidores, propagação de informação e re-encaminhamento de pedidos para os servidores finais. A estrutura organizacional dos sistemas *BigTable* permite igualmente escalar eficientemente com a adição de novos servidores com impacto mínimo no desempenho. No entanto a utilização de serviços como localização de servidores (p.e. serviço *chubby*) ou componentes para gerir o estado do serviço (i.e. servidor *master* poderão ser pontos críticos de falha impossibilitando o uso do sistema. Relativamente ao *MongoDB*, a capacidade de escalar depende do rácio escolhido entre clientes e servidores *mongos*, que não sendo equilibrado poderá causar engarrafamento dos clientes nestes componentes. Adicionalmente é necessário inferir qual o número de *shards* e servidores *mongod* a disponibilizar que maximiza a distribuição da carga. Tal como o *BigTable*, depende de um serviço de gestão do sistema.

A implementação de serviços que apresentam padrões de acessos focado nas escritas sugerem a utilizam de sistemas como o *BigTable*. Estes sistemas optimizam as escritas implementado um mecanismo de *log* sequencial em disco. O *MongoDB* utiliza mecanismos de indexação, actualizados na inserção de dados, para melhorar o desempenho das leituras. O *Dynamo* está orientado para aplicações que necessitam alta disponibilidade onde as escritas não são rejeitadas mesmo na existências de falhas na rede ou no sistema,

mantendo o nível de desempenho. O *Cassandra* implementa o melhor dos dois mundos: a estrutura eficiente do *Dynamo* e o modelo de dados do *BigTable*, o que permite criar um serviço disponível, robusto e de alto desempenho com um modelo de dados mais rico e adaptável a um maior número de aplicações. Na tabela 2.2 está apresentado um resumo sobre repositórios discutidos nesta secção.

2.4 Transformação Operacional

No domínio do sistemas distribuídos a imposição de garantias de consistência é um dos maiores problemas a abordar durante o desenvolvimento, criando desafios ao nível do desenho e implementação. Uma nova técnica baseada em transformação operacional foi proposta no contexto da edição colaborativa, com o objectivo de garantir consistência entre réplicas sem sacrificar a rapidez de reposta e o nível de concorrência entre participante [EG89]. Nestes sistemas o estado das réplicas é composto por uma sequência de caracteres, podendo ser executadas duas operações: $insert(pos, ch)$, para inserir o caracter ch na posição pos; $delete(pos)$, para remover o caracter presente na posição pos.

As técnicas de transformação operacional têm por base três propriedades essenciais: convergência, preservação da ordem causal das operações e preservação da intenção das operações. Um sistema converge quanto todas a réplicas convergem para um estado final comum. Se todas as réplicas executarem o mesmo conjunto de operações sobre o mesmo estado inicial então o estado final alcançado será o mesmo. As operações são ordenadas segundo uma relação causal de forma a capturar o efeito que a execução de uma operação tem na geração de uma operação posterior. Isto é, se execução de uma operação, gerada previamente, precede a geração de uma outra na mesma réplica então esta ordem deve ser preservada em todas as outras réplicas. Apenas utilizando estas duas ultimas propriedades juntamente com a técnica de transformação de operações é possível a um sistema obter consistência. A intenção de uma operação é descrita pelo efeito que a acção desencadeada pela operação produz no estado do sistema, ou seja, que: um $delete$ remove em todas as réplicas o caracter que for removido na réplica onde a operação foi executada originalmente; a posição relativa de um novo caracter adicionado por uma operação de $insert$ face a todos os outros caracteres do texto é a mesma em todas as réplicas. Na presença de conflitos entre operações é preciso garantir que estas são manipuladas de forma a preservar a intenção de cada no estado do sistema. O leitor pode encontrar a prova para a correcção do sistema em [EG89, SE98].

Uma primeira aproximação foi proposta no algoritmo *dOPT* pela equipa do sistema *Grove*, que apenas oferece garantias de convergência da réplicas ordenando causalmente as operações. Posteriormente foi verificado que o sistema anterior introduzia um novo tipo de violação de consistência: violação da intenção das operações. Este problema foi corrigido num novo algoritmo, *GOTO*, através da criação das propriedades TP1 e TP2 [SE98].

3

Desenho do sistema MixCloud

Neste capítulo apresenta-se o modelo de dados e a API genérica do sistema MixCloud, seguido da sua arquitectura. Apresentar-se-á um modelo que utiliza a transformação de operações para garantir uma consistência eventual através do estabelecimento de uma ordem total de operações em cada réplica. Por fim, serão apresentadas as funcionalidades dos componentes que formam o sistema e como estas conseguem garantir as propriedades acima indicadas.

O sistema MixCloud pretende fornecer um serviço robusto e eficiente de armazenamento de dados para uma infra-estrutura de *clusters* ou de *cloud*. Para tal, combina e integra um conjunto de repositórios chave/valor. Sobre estes repositórios é implementado um mecanismo de replicação de forma a tornar o serviço disponível e tolerante a falhas nos repositórios. O desempenho pode também ser melhorado pela distribuição da carga pelos repositórios.

3.1 Modelo de Dados

O MixCloud adopta um modelo de dados simples com uma estrutura flexível, isto é, adaptável às diferentes necessidades das aplicações e mapeável sobre qualquer repositório, baseado no modelo proposto no sistema BigTable [CDG+08]. O modelo é composto por vários níveis de organização: atributos, famílias e linhas. Um atributo funciona como a unidade básica de acesso que indexa os valores do sistema. As famílias são formadas por conjuntos de atributos. Adicionalmente cada família contém um número arbitrário de linhas que estão associadas a esses atributos. Adoptamos este esquema dinâmico para oferecer ao cliente uma maior liberdade na organização dos seus dados e uma melhor adaptação às diferentes necessidades que as aplicações possam ter. Na tabela 3.1

	Família Um			Família Dois	
	A_1	A_2	A_3	A_4	A_5
L_1	V_1	V_2	V_3		
L_2				V_4	V_5
L_3	V_6	V_7		V_9	V_10

Tabela 3.1: Exemplo de modelo de dados

encontra-se um exemplo de uso do modelo que propomos, na qual os dados estão organizados em tabelas compostas por conjuntos de atributos e indexados por uma chave, tal como numa base de dados relacional. A novidade está na introdução do elemento família, que permite agregar conjuntos de atributos que se encontrem associados num mesmo domínio.

Um atributo é constituído por um nome que o identifica e um valor que representa os dados introduzidos pelos clientes. O seu nome consiste num conjunto de caracteres e o valor associado a ele é um conjunto de *bytes*. No nosso exemplo os atributos correspondem à sigla A, concretamente A_1, A_2, A_3, A_4 e A_5. Por sua vez os atributos estão divididos pelas famílias Um e Dois. As famílias são intencionalmente pequenas em número e não podem ser criadas ou removidas durante a operação do sistema. Estas são identificadas por um nome único. No entanto, famílias distintas podem conter atributos com o mesmo nome. As linhas são indexadas e ordenadas por uma chave única, representada por um conjunto de caracteres, que lhe está associada. Seguindo o exemplo, o conjunto das linhas é formado pelos elementos L_1, L_2 e L_3. A primeira, L_1, está associada à família Um e aos atributos A_1, A_2 e A_3. L_2 está associado à família Dois e aos atributos A_4 e A_5. Por fim, L_3 encontra-se em ambas as famílias, com atributos A_1, A_2, A_4 e A_5. Note-se que uma linha pode estar contida em mais que uma família.

Cada linha pode conter um conjunto de diferentes atributos. Ao contrário das famílias, este esquema não é pré-definido, permitindo a criação e remoção para além da alteração de linhas e atributos.

Todos os valores armazenados nas tabelas são identificados por uma chave, composta pelos elementos linha, família e atributo. Por exemplo o valor V_1 é identificado pela chave L_1 *: Família Um :* A_1, enquanto o valor V_9 é identificado pela chave L_3 *: Família Dois :* A_4. Este modelo pode ser adaptado ao modelo chave/valor na medida em que a chave é composta pelos identificadores família, linha e atributo que indicam onde o valor se encontra guardado. As leituras sobre uma linha numa família são atómicas independentemente do número de atributos acedidos.

Como consequência, este modelo pode ser facilmente adaptado aos modelos de dados dos repositórios que apresentam estruturas e elementos diferentes. Um exemplo é o MongoDB, onde os documentos, os seus campos e as colecções podem ser correspondidas às linhas, às colunas e às famílias, respectivamente.

3.2 MixCloud API

A API do sistema MixCloud foi criada tendo por base a API genérica proposta por Brian F. Cooper et al. na ferramenta de *benchmark* para repositórios na *cloud* YCSB [CST+10]. As operações que a constituem dividem-se em escritas – *Put* e *Delete* – e leituras – *Read* e *Scan*. As operações de escrita alteram os valores associados às chaves introduzidas no sistema enquanto as leituras apenas retornam os valores referenciados pelas chaves indicadas pelo cliente. Todas as operações são atómicas dentro de uma linha quanto a escritas concorrentes. A API é apresentada na tabela 3.2, com os respectivos argumentos. Esta é genérica e independente da API adoptada pelos repositórios usados no sistema, permitindo esconder da aplicação os detalhes do modelo de dados e organização destes repositórios.

As operações de escrita são utilizadas para alterar os dados no sistema. A adição de uma nova chave e valor dentro de uma família é feita através de uma operação *Put* e a eliminação de um atributo e respectivo valor de uma chave existente, o *Delete*. Como descrito na tabela 3.2, a operação *Put* recebe em argumento os elementos família, linha e o conjunto de pares contendo o nome do atributo e o valor respectivo a inserir/actualizar na linha. Esta operação apresenta o seguinte comportamento: se a linha indicada não existir, cria-se uma nova linha e insere-se todos os atributos e respectivos valores; caso contrário insere-se apenas os atributos que ainda não tenham sido criados e altera-se os valores para os atributos já existentes. Por sua vez, a operação *Delete* recebe como argumentos a família, a chave e os atributos a eliminar. O *Delete* funciona do seguinte modo: remove-se os atributos indicados em argumento e por fim, no caso da linha ser constituída por um conjunto de atributos vazios, esta é eliminada. No caso de uma operação ter sido executada com sucesso garante-se que as alterações produzidas são mantidas persistentes nos repositórios. Na presença de falha, não é realizado um *rollback* e as alterações já produzidas nos repositórios até esse ponto são mantidas.

A operação *Read* lê conjuntos de atributos e os seus valores sobre uma linha e família, especificados no argumento. Por outro lado, a operação *Scan* permite iterar por ordem lexicográfica dentro de uma família sobre um conjunto de linhas, dados a chave da linha inicial e o número de linhas a iterar, lendo os atributos indicados que estão associados a cada linha desse conjunto. Esta última operação pode ser vista como um agregado de *Reads* dentro de uma família, onde é feita uma leitura por linha, a um conjunto de atributos pré-definido, até a um limite de número de linhas. O resultado corresponde ao agregado dessas leituras, onde cada leitura está separada pelo nome da linha correspondente. Ao não especificar as linhas ou os atributos, é lida por inteiro a família em argumento. As operações de leitura executam sobre um estado que será pelo menos igual ao estado do servidor aquando da última operação de leitura ou escrita. Assim o sistema implementa garantias de sessão *"monotonic reads"* e *"read your writes"* [TDP+94].

Operação	Resultado
put(família, linha, <atributos,valores>)	-
delete(família, linha, atributos)	-
read(família, linha, atributos)	Valor dos atributos
scan(família, linha inicial de pesquisa, #linhas, atributos)	Conjunto com valores dos atributos de cada linha

Tabela 3.2: API do sistema MixCloud.

3.3 Arquitectura do Sistema

O sistema MixCloud implementa um serviço de armazenamento com base num *middleware* suportado por dois componentes principais: *ClientProxy* e *ServerProxy*, como apresentada na figura 3.1. Localizado no *front-end* está o *ClientProxy*, permitindo que os clientes comuniquem com os repositórios. O seu objectivo passa por gerir todo o processo de tratamento das operações dos clientes de forma transparente para estes. No *back-end* do sistema, encontram-se os *ServerProxies* que estabelecem a ligação entre os vários *ClientProxies* e os repositórios de dados a que estão directamente relacionados. Pretende-se que este componente ofereça uma plataforma que permita executar correctamente as operações recebidas, respeitando as garantias que a API oferece, e estabelecer o modelo de consistência eventual entre réplicas subjacentes ao serviço.

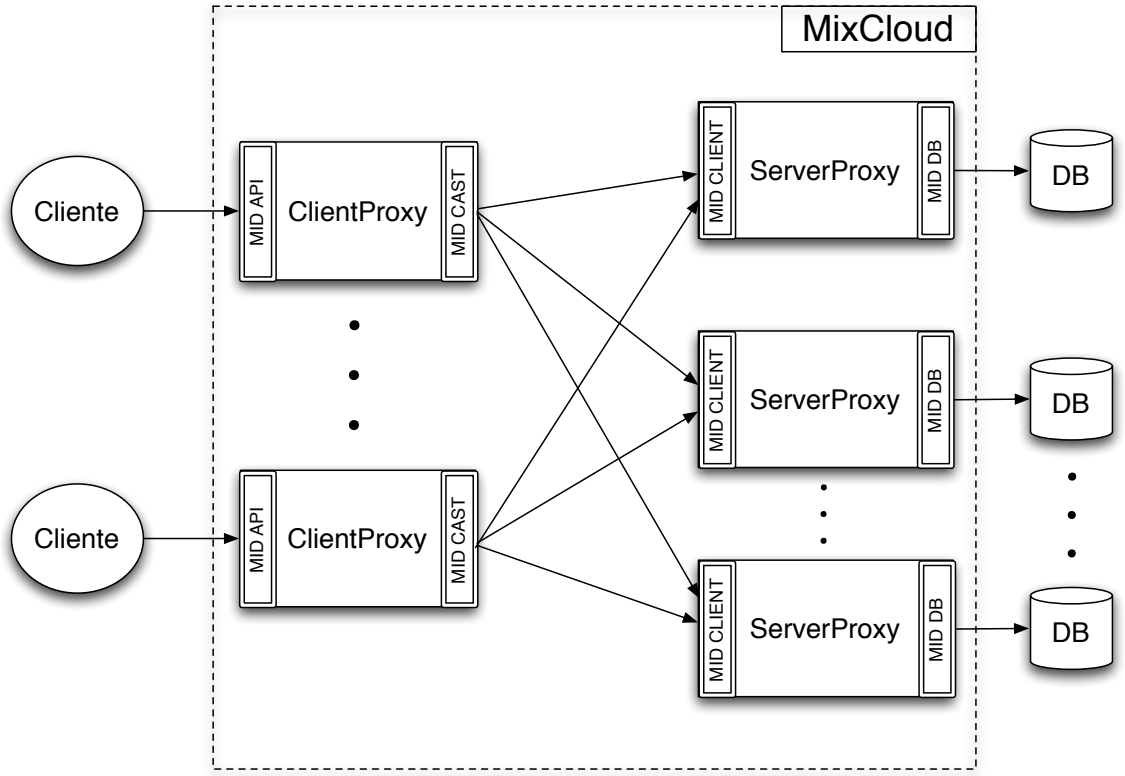

Figura 3.1: Arquitectura do MixCloud.

O componente *ClientProxy* representa o cliente perante o serviço MixCloud e os seus *ServerProxies*. Funciona como um *proxy* que recebe as operações vindas do cliente e envia-as para os vários repositórios a fim de serem executas, esperando pelas respostas a entregar à aplicação. Gere também todo o processo de comunicação com os servidores, desde o estabelecimento dos canais de comunicação com cada *ServerProxy* até à codificação das operações a enviar. A figura 3.2 apresenta a estrutura interna do cliente, formada pelos componentes *proxy*, *service*, *time* e *transport*. O *proxy* funciona como ponto de entrada no sistema para aplicação do cliente através da *MID API*, que implementa a MixCloud API descrita na secção 3.2. No componente *service* é feita a gestão do serviço em termos do tratamento das operações e das respectivas respostas. Este trabalha em conjunto com o componente *time* que gere o tempo no cliente. O componente *transport* controla as comunicações entre os clientes e o conjunto dos *ServerProxies*. Implementa a biblioteca *multicast*, representada na figura pelo elemento *MID CAST*, que suporta canais de comunicação fiáveis estabelecidos com cada servidor. As operações são enviadas em paralelo para os servidores segundo a ordem FIFO para garantir a ordem das operações do cliente. O cliente MixCloud oferece transparência ao cliente, escondendo como os sistemas e os dados estão implementados e onde se encontram localizados.

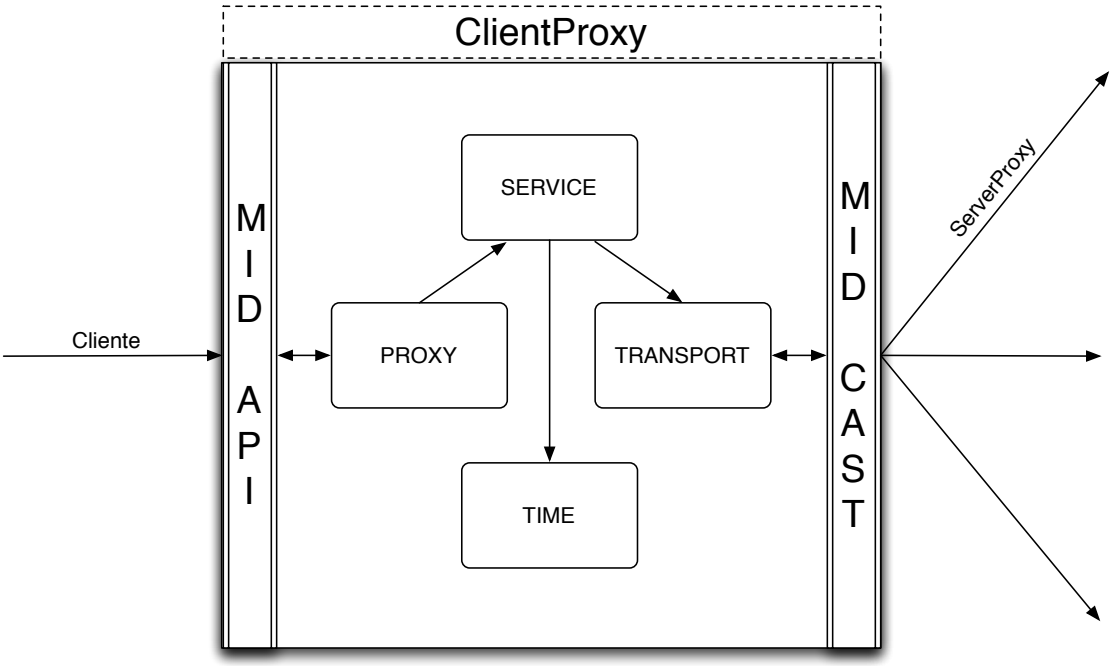

Figura 3.2: Desenho do *ClientProxy*.

Uma aplicação envia operações para o serviço MixCloud utilizando a API do serviço, através do componente *proxy*. A operação é de seguida entregue ao componente *service*

que encaminha-a para o componente *transport* para o envio aos servidores. Este componente espera pelas respostas a entregar à aplicação.

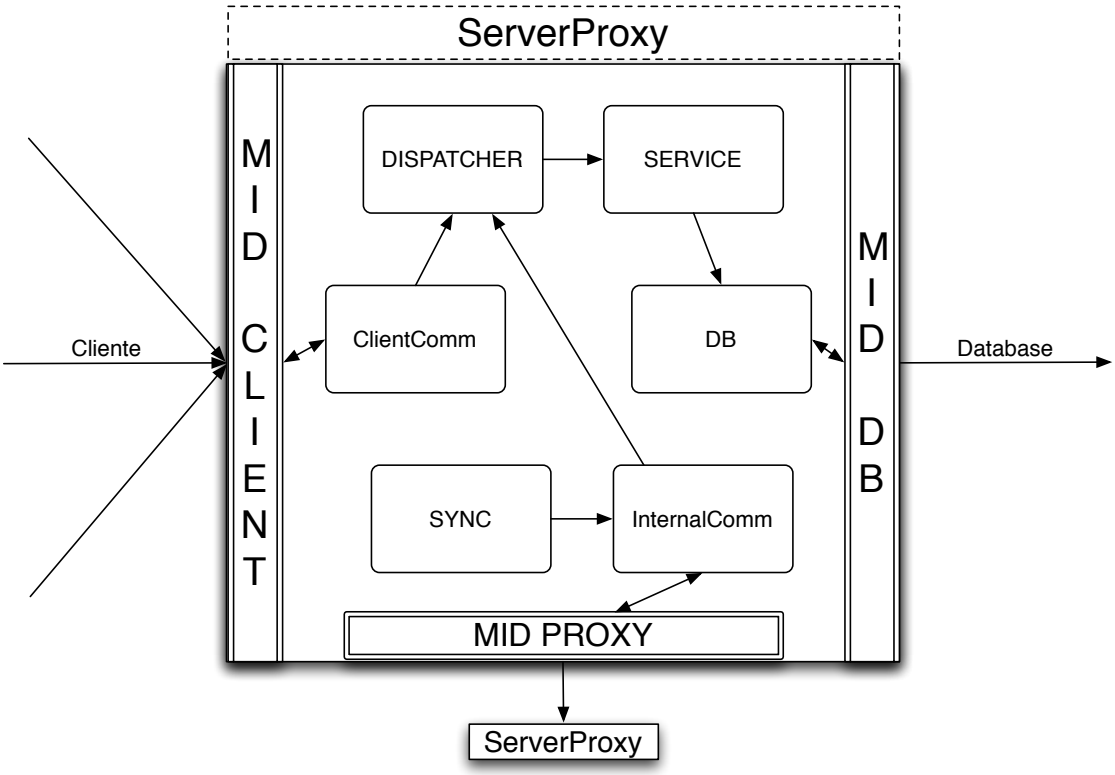

Figura 3.3: Desenho do *ServerProxy*.

A estrutura interna do *ServerProxy* (figura 3.3) é constituída pelos componentes *client comm*, *dispatcher*, *service*, *db*, *sync* e *internal comm*. Verifica-se que o *ServerProxy* apresenta uma maior complexidade estrutural relativamente ao *ClientProxy* pois é neste componente que se implementa o núcleo funcional do serviço que permite manipular os dados e manter um visão coerente dos mesmos. As comunicações externas com os clientes são geridas pelo componente *client comm*, que aceita novas conexões, recebe operações e devolve as respostas. O componente *dispatcher* é o elemento central, na medida em que controla quais os passos a efectuar desde que uma operação é recebida no *proxy* até ser executada no repositório (*database*) e de seguida recolhe a resposta a retornar ao cliente. As garantias associadas ao serviço estão implementadas no *service*. A ligação entre o *ServerProxy* e o repositório é estabelecida pelo componente *db*. Este componente converte as operações para o formato do modelo de dados e operações do repositório. Relativamente ao mecanismo de sincronização, foi acrescentado um componente *sync* para gerir este processo e garantir a convergência das réplicas. O *internal comm* estabelece a comunicação com outros *proxies* para verificação e actualização do estado. Este componente pode vir a incluir novas funcionalidades como troca de informação para formular estatísticas ou implementação de um mecanismo de *"failover"* com detecção de falha de um

outro *proxy*.

Cada um dos servidores mantém um vector versão que representa o seu estado corrente, localizado no componente *sync*. As posições deste vector versão contêm o valor do relógio lógico da última operação efectuada pelos clientes, sendo que cada posição corresponde a um cliente do sistema e é indexada pelo seu identificador. Após a execução de uma operação vinda de um cliente, a entrada respectiva do vector versão é actualizada com o valor máximo entre o seu valor e o relógio da operação. O vector permite detectar divergências de estado entre servidores relativamente ao conjunto de operações em falta. Suponha que existem os vectores versão VV_1 e VV_2 de servidores diferentes, S_1 e S_2, respectivamente. São detectadas operações em falta para algum cliente i se $VV_1[i] < VV_2[i]$ em S_1 ou $VV_2[i] < VV_1[i]$ em S_2. O conjunto de operações em falta contém um relógio que se enquadra entre os valores de $VV_1[i]$ e $VV_2[i]$.

As operações vindas dos clientes chegam a um servidor através do componente *client comm*, que envia para o componente *dispatcher*. Este componente, por seu lado entrega-as ao componente *service*, garantindo-se que a execução da operação não viola as garantias do serviço. Por fim, as operações são enviadas para o componente *db* para serem enviadas e executadas nos repositório. O componente *sync* periodicamente activa a fase de sincronização e com auxílio do componente *internal comm* comunica com um outro servidor de forma a trocarem o conjunto de operações em falta no estado de um com o outro.

A adopção desta estrutura de clientes e servidores permite distribuir o serviço pelos diferentes componentes. Cada aplicação do cliente possui uma ligação directa com o *ClientProxy*, pelo qual acede ao serviço MixCloud. Apenas um *ClientProxy* fica afectado a uma aplicação, de forma a gerir toda o processo de execução de operações e comunicações com os servidores. O *overhead* do processo de gestão fica dividido por todos os clientes, tendo um menor impacto no desempenho geral do sistema. No lado dos servidores, a utilização de apenas um *ServerProxy* por repositório permite que haja controle sobre o estado actual de cada réplica armazenada nesse repositório. Simplifica assim o mecanismo de coerência das réplicas nos diferentes repositórios. Além disso, permite distribuir a carga e aumentar a disponibilidade do serviço mesmo que algum dos *proxy* ou repositório apresente falhas e deixe de funcionar.

O desenho do *MixCloud* apresentado está simplificado para melhor compreensão do leitor. Caso necessário, os *ServerProxies* podem ser agregados em grupos, à semelhança dos *shards*, e cada grupo ficará associado a um repositório. No interior de um *shard* adopta-se um esquema baseado em *master-slave*, onde um dos servidores funciona como responsável do grupo e os restantes como *slaves*. Os clientes possuem conhecimentos dos elementos de cada grupo, podendo escolher qual dos servidores contactar. Esta solução baseia-se no sistema MongoDB e pode ser encontrada em [Sha11]. Esta aproximação permite eliminar o ponto de falha único existente nos *ServerProxies* e por outro lado distribuir a carga pelos servidores do *shard* para melhorar o desempenho. Esta solução não se encontra implementada.

Cada servidor aplica as operações pela ordem de chegada sem consenso prévio. Devido à latência das comunicações entre o cliente e os servidores, estes podem divergir no estado final ao receberem o conjunto de operações por ordens diferentes. Para garantir consistência adopta-se uma aproximação baseada em transformação operacional que permite os servidores executarem as operações por ordens equivalentes e assim alcançar um estado final das réplicas consistente. Por esta via torna-se possível o sistema implementar um modelo de consistência eventual.

3.4 Transformação Operacional

Neste trabalho, ao contrário das soluções de transformação de operação usadas na edição cooperativa de texto [EG89, SE98], o objectivo é apenas garantir a convergência dos dados. Adicionalmente, o modelo de dados torna o problema bastante mais simples, pois as operações usam identificadores únicos imutáveis para identificar os dados que são alterados.

Propriedade de Convergência

A execução de um conjunto de operações em dois servidores diferentes podem resultar em estados finais diferentes se a ordem não for igual. No caso do conjunto de operações ser totalmente ordenado, pode-se provar por indução que a convergência é garantida. Vamos apresentar uma definição de ordem total e a de relógio lógico segundo Leslie Lamport [Lam78], que permite ordenar totalmente um conjunto de operações.

Definição 1. *Convergência.*

Um sistema converge quando todos os servidores vêem o mesmo conjunto de operações e cada um atinge o mesmo estado final após executar todas as operações do conjunto partindo do mesmo estado inicial.

Definição 2. *Relógios lógicos.*

Um relógio lógico consiste num valor que cresce cada vez que uma nova operação é gerada num cliente. Mais especificamente, definimos um relógio lógico C_i no cliente i como sendo uma função monotonamente crescente que atribui um número $C_i(O_i)$ a qualquer operação O_i.

Definição 3. *Ordem Total*

No domínio da teoria de conjuntos, uma ordem total é uma relação binária (denotada por $\leq$) definida sobre um conjunto X, verificando as seguinte propriedades para todo a, b, c elementos de X:

1. *Se $a \leq b$ e $b \leq a$, então $a = b$ (antissimetria)*
2. *Se $a \leq b$ e $b \leq c$, então $a \leq c$ (transitividade)*
3. *$a \leq b$ ou $b \leq a$ (totalidade)*

Definição 4. *Relação de ordem total "$\Rightarrow$".*

Considere-se as operações O_i e O_j geradas no cliente i e j respectivamente. Cada cliente tem associado um ID único com o seu nome, i para o client i e j para o cliente j. $O_i \Rightarrow O_j$, sse $C_i(O_i) < C_j(O_j) \vee (C_i(O_i) = C_j(O_j) \wedge i < j)$.

Esta última definição estabelece então um método para ordenação de operações que, portanto, permite munir o sistema da propriedade de convergência.

Algoritmo

As operações recebidas no cliente MixCloud vindas do cliente são propagadas em paralelo para os servidores. No lado dos servidores, o conjunto de operações é recebido por ordens diferentes. Os estados dos servidores podem divergir se cada um aplicar o conjunto de operações que conhece. O envio das operações do cliente para os servidores é realizada segundo uma ordem FIFO, sendo enviado uma operação de cada vez e esperada a respectiva resposta. Assim os servidores conseguem ordenar as operações vindas de um cliente e garantir que a sua próxima operação é executada sobre um estado coerente que reflecte as operações anteriormente emitidas por esse cliente.

Consideremos, por simplicidade, o caso em que cada operação de escrita, *Put* e *Delete*, realiza alterações no estado dos servidores e acede apenas a uma chave. Cada operação tem uma estampilha associada na forma $(relógio\_lógico,\ id\_cliente)$ que permite estabelecer uma relação de ordem total. Uma possível solução para garantir convergência seria utilizar uma aproximação *"last-write-wins"*. Para a sua implementação apenas é preciso guardar para cada chave a estampilha da última operação executada no servidor. Se num qualquer servidor chega uma operação gerada no cliente i e a sua estampilha é maior que a estampilha da última operação executada, vinda de cliente j, então vai ser executada. Caso contrário, a operação é ignorada.

No entanto, o MixCloud implementa operações (leitura e escrita) que acedem a mais que uma chave num dado momento. Como tal esta solução necessita uma extensão para armazenar a estampilha da última operação executada para cada chave acedida. Considerando apenas as escritas, seria possível verificar quais as colunas que seria necessário alterar consultando as estampilhas temporais da operação recebida e de cada uma das chaves. Para suportar operações mais complexas [1], implementámos uma solução baseada em transformação de operações. Esta opção permite evitar o overhead de armazenar toda informação referente às estampilhas temporais e aumenta a rapidez de execução das operações. Igualmente, permite que operações vindas fora de ordem nos servidores possam ser executas após transformação.

A nossa solução é composta por um esquema de relógios lógicos – para ordenar totalmente as operações – e um método de transformação.

[1] Por exemplo, operações para as quais não seja possível determinar no cliente quais as chaves acedidas - e.g. atribuir um determinado valor a todas as linhas que satisfazem um dada condição.

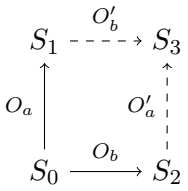

Figura 3.4: Diagrama de Equivalência da Transformação - Começando no estado inicial S_0 e executando a operação O_a (aresta sólida à esquerda) obtém-se o estado intermédio S_1. Em seguida, a operação O'_b é executada (aresta superior a traço interrompido) em S_1 e o estado S_3 é alcançado. Por outro lado, o estado intermédio S_2 é obtido pela execução de O_b no estado S_0 e finalmente o estado S_3 aparece após a execução da operação O'_a em S_2. Conclui-se que é possível convergir para um estado final partindo de um estado inicial comum mesmo que os estados intermédios sejam diferentes (neste caso, S_1 e S_2).

Usando um método de transformação T garante-se que o resultado final é igual ao da execução das operações pela ordem total definida. O método T satisfaz a seguinte propriedade de equivalência: para duas operações, tal que, $O'_a = T(O_a, O_b)$ e $O'_b = T(O_b, O_a)$, o resultado de executar $O_a \circ O'_b$ ou $O_b \circ O'_a$ ($\circ$ denota a composição de operações) no estado inicial de um objecto traduz-se num mesmo estado final (denotado como $O_a \circ O'_b \equiv O_b \circ O'_a$, retratado em 3.4).

Cada servidor mantém um *log* com as operações executadas localmente. Quando uma operação é recebida vinda de um cliente, o algoritmo em 3.1 é executado. O algoritmo começa com a criação de uma variável EO para armazenar o resultado final da transformação desta operação. A operação passa por uma fase de ordenação, onde é encontrada a sua posição na ordem total e, por conseguinte, determinado o conjunto das operações que se encontram em posições posteriores.

Algoritmo 3.1 Algoritmo para garantir a relação de ordem total e convergência

Inicialização.
$log \leftarrow vazio$

Ao receber a operação O vinda de um cliente
$EO \leftarrow O$
$j \leftarrow 0$
$pos \leftarrow Order(O, log)$
while $j < log.size() \wedge EO \neq NoOp$ **do**
  $EO \leftarrow T(EO, log[j])$
  $j \leftarrow j + 1$
end while
if $EO = NoOp$ **then**
  nenhuma acção é realizada
else
  executar EO
end if

Algoritmo 3.2 Inserir uma nova operação no *log* mantendo a relação de ordem total entre operações

$Order(O, log) : pos$, posição onde a operação foi inserida
$pos \leftarrow log.size()$
while $pos > 0 \wedge O \Rightarrow log[pos]$ **do**
  $pos \leftarrow pos - 1$
end while
retornar pos

O objectivo é garantir que as modificações já realizadas pelas operações desse conjunto se mantenham e não sejam eliminadas pela execução da operação recebida, que está fora de ordem. Em seguida, passamos para a fase de transformação que elimina da operação recebida as chaves comuns com todas as operações desse conjunto. A operação recebida é assim transformada, contendo apenas as chaves que não foram modificadas por operações posteriores a si. Se o conjunto de chaves da operação transformada for não vazio, a operação é então inserida no *log* segundo a ordem total e posta a executar no repositório. Caso contrário, chamamos a esta operação $NoOp$ e nada é feito.

O método *Order* (especificado em 3.2) consiste basicamente em varrer o *log* desde a ultima operação até à primeira e encontrar qual é a posição onde a operação se insere respeitando a ordem total. Para tal, passo a passo, são comparados o relógio lógico da operação recebida com os das operações no *log*. No fim a posição encontrada é retornada.

O método de transformação T (especificado em 3.3) utiliza a posição obtida pelo método *Order* para transformar *iterativamente* a operação recebida tendo em conta as operações que se encontram no *log* após a posição indicada. A cada iteração do algoritmo, o conjunto de chaves da operação recebida e o da operação na posição seguinte são comparados, de forma a extrair o subconjunto de chaves a aceder comum às duas. Este subconjunto de chaves é eliminado na operação recebida.

Por simplicidade de escrita, serão introduzidas algumas notações úteis sobre propriedades das operações.

Definição 5. *Propriedades das operações.*

Sejam $O = \langle (k_1, v_1),\ (k_2, v_2),\ ...,\ (k_m, v_m) \rangle$ *e* $O' = \langle (k'_1, v'_1),\ (k'_2, v'_2),\ ...,\ (k'_n, v'_n) \rangle$. *Definimos que:*

(i) $O.tuplo = \{(k_i, v_i),\ i = 1, ..., m\}$;

(ii) $O.chaves = \{k_i,\ i = 1, ..., m\}$;

(iii) $O \cap O' = \{k \in O.chaves \wedge k \in O'.chaves\}$;

(iv) $O \backslash O' = \langle (k''_1, v''_1),\ ...,\ (k''_l, v''_l) \rangle$ *tal que* $(O \backslash O').chaves = O.chaves \backslash O'.chaves$.

Nesta secção foi apresenta um algoritmo de transformação operacional e propriedades associadas para garantir consistência num ambiente distribuído. Ao contrário do *dOPT* e *GOTO*, o nosso algoritmo apenas garante convergência relativamente a uma ordenação total das operações.

Algoritmo 3.3 Método de transformação T

$T(O, O_k) : EO$, operação transformada
$EO \leftarrow O$
if $EO \cap O_k$ **then**
  $EO \leftarrow EO \backslash O_k$
end if
retorna EO

Integração no MixCloud

Os relógios apresentados na subsecção anteriores são utilizados no MixCloud para controlar a evolução do sistema em termos das operações enviadas pelos clientes e operações executadas nos servidores. No lado dos servidores, os relógios contidos nas operações vindas dos clientes são agregados nos vectores versão que representam o estado, como indicado na secção 3.3. Estes vectores são disseminados pelos clientes e integrados no vector versão local para que possam ter uma visão global do estado do sistema. O relógio local do cliente é actualizado com base no valor máximo de todas as posições do vector versão resultante do passo anterior, permitindo que o seu relógio acompanhe o passo do relógio dos restantes clientes. Assim, é possível que as operações dos vários clientes se aproximem da ordem real. O vector local de cada cliente é disseminado pelos vários servidores em cada operação enviada, para que estes possam comparar o seu estado com o estado geral e activar o processo de sincronização caso seja necessário. Por esta via conseguimos acelerar o processo de convergência dos estados dos servidores e permitir ao sistema atingir as garantias de sessão indicadas anteriormente.

3.5 Funcionamento do Sistema

Os utilizadores comunicam com o MixCloud via aplicação cliente que suporta a API interna do sistema. Seguindo o fluxo das operações no cliente (figura 3.5), a aplicação envia novas operações para o servidor através do cliente do MixCloud, *ClientProxy*, que utiliza o componente *proxy* como ponto de entrada. O componente *Service* ao receber uma nova operação, acrescenta-lhe uma estampilha temporal, o vector versão local e um número de sequência. O número de sequência consiste num contador, incrementado a cada nova operação, que identifica internamente cada operação do cliente. A operação é então codificada e preparada para ser enviada para os servidores. O envio da operação é realizado em paralelo para todos os servidores através do componente *transport*, que espera sincronamente pelas respostas.

Ao receber uma resposta, o cliente procede à actualização do seu relógio guardado em *time*. Apenas a primeira resposta recebida é entregue à aplicação, via componente *service*, enquanto as restantes são descartadas. Esta aproximação permite aumentar o desempenho tirando partido do repositório mais rápido a responder para esta operação. Cada

resposta contém um vector versão representando o estado do servidor. Os clientes também mantêm um vector versão com o máximo dos vectores recebidos, cujo o resultado consiste no conjunto dos valores máximos para cada uma das posições desses vectores. Por sua vez, o novo valor do relógio é calculado a partir do vector resultante, como indicado na secção 3.4.

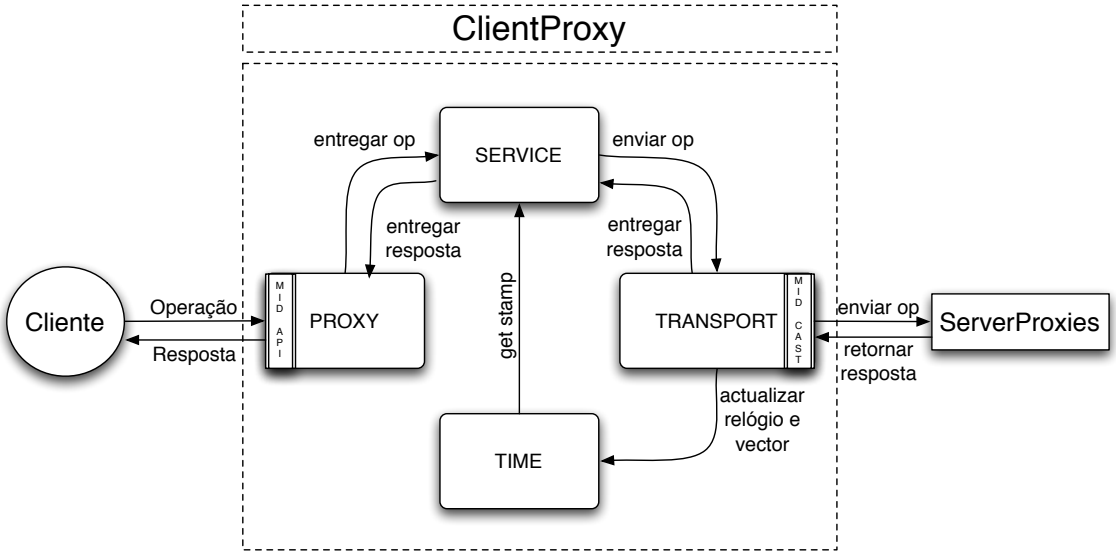

Figura 3.5: Desenho interno do *ClientProxy*.

As operações de escrita são sempre difundidas para todos os servidores, enquanto as de leitura são enviadas apenas para o servidor conhecido mais rápido num dado momento, beneficiando de uma melhor distribuição de trabalho. Ao usar esta aproximação é possível melhorar o tempo de resposta médio do serviço, pois os servidores conseguem responder imediatamente e de forma independente sem ter de esperar pela sincronização.

Seguindo o fluxo das operações no servidor apresentado na figura 3.6, este recebe operações vindas de vários clientes distribuídos através de um único ponto de entrada utilizando o componente *clientComm*. Este componente suporta concorrência nos acessos e aceita múltiplas conexões em simultâneo. Após receber uma nova operação, o *clientComm* coloca-a numa fila de operações em espera. O componente *dispatcher* por sua vez retira as operações da fila pela ordem de chegada e executa o protocolo de tratamento da operação consoante o seu tipo. Se for uma escrita é entregue ao *service* para ser ordenada e transformada relativamente ao *log* de operações. No caso de uma leitura, é directamente enviada para o repositório.

A transformação de operações de escrita elimina inconsistências que a execução da operação na forma original poderia causar no estado da réplica. A este nível operações podem ser transformadas concorrentemente, desde que as chaves não pertençam

à mesma família e linha. Caso contrário o *log* poderia apresentar inconsistências e a dependência entre operações que acedem à mesma família e linha perder-se-ia [2]. Torna-se assim possível garantir que as réplicas convergem para um mesmo estado, ainda que as operações tenham sido aplicadas por ordens diferentes. A operação final transformada é retornada de novo para o *dispatcher*.

Tanto as escritas como as leituras são enviadas para o repositório através do componente *db*. As respostas recebidas são propagadas pelo caminho inverso – *dispatcher* e em seguida *internalComm* – até alcançar o cliente. O *ServerProxy* após executar correctamente uma operação de escrita actualiza o seu vector versão na posição correspondente ao cliente. Cada resposta tem associado o vector versão do servidor.

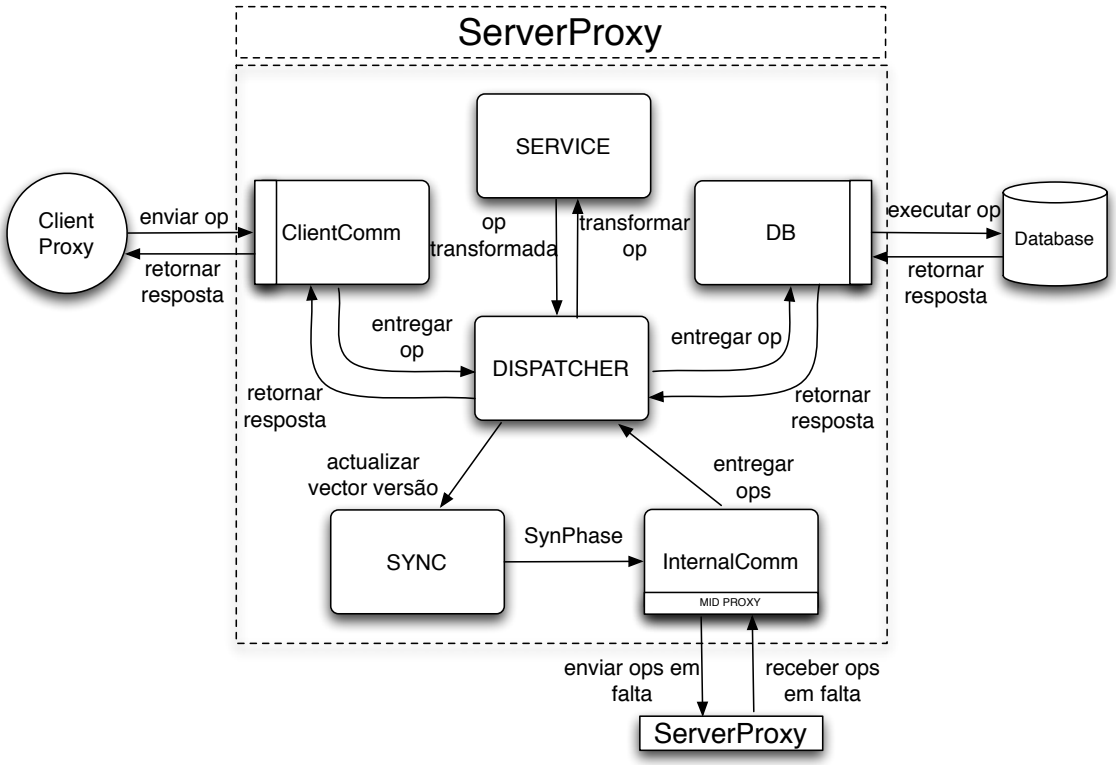

Figura 3.6: Desenho interno do *ServerProxy*.

As operações de leitura suportadas pelo serviço do MixCloud oferecem garantias de sessão *"monotonic reads"* e *"read your writes"*. Antes de executar estas operações, é necessário verificar se as escritas de qual a leitura depende (escritas previamente executadas por um mesmo cliente) se encontram reflectidas no vector versão da replica. Se existirem operações de escrita em falta, deve-se esperar pela sua chegada. No caso normal, a operação pode ser imediatamente executada.

Foi adicionada uma fase de sincronização para acelerar o processo de convergência

[2]Considerando duas operações O_1 e O_2, respectivamente com estampilha E_1 e E_2 em que $E_2 < E_1$, vem que a transformação de O_2 depende da transformada de O_1 e não de O_1. A execução simultânea das duas transformações pode assim causar incoerência no resultado da transformação de O_2.

de estado entre as réplicas e permitir a tolerância a falhas de um cliente. O componente *sync* encontra-se responsável por gerir a periodicidade com que a sincronização ocorre. Ao alcançar um novo período, o *sync* acciona a fase de sincronização e emite um pedido ao *internalComm* para iniciar a troca de informação com um outro *ServerProxy*, escolhido segundo uma métrica pré-definida. Numa primeira fase, os servidores trocam os seus vectores versão e cada um efectua a comparação entre o seu vector local e o vector recebido de forma a calcular o conjunto das operações em falta no seu estado, como descrito na secção 3.3. O conjunto anterior é enviado para o outro servidor como pedido das operações a receber. Do outro lado, o *proxy* reencaminha o pedido para o *service* que colecciona as listas de operações em falta por cliente. Essas listas são enviadas de volta ao emissor do pedido. As operações recebidas pela sincronização seguem o mesmo percurso das operações vindas dos clientes, isto é, passam por uma ordenação e transformação prévia e só depois são executadas. Este processo garante que a propriedade de convergência é mantida respeitando as dependências entre operações. O sistema suporta uma qualquer métrica de escolha de um *ServerProxy*, podendo ser simplesmente aleatório ou então segundo um esquema *"round-robin"*.

O sistema MixCloud está orientado para repositórios de dados que não oferecem de raiz soluções de replicação de dados. Ao agregarmos diferentes sistemas podemos assim explorar melhores desempenhos e o aumento da disponibilidade para o serviço MixCloud. O grau de independência entre os repositórios permite explorar o rácio ideal entre os servidores e os clientes.

4

Implementação

Tal como foi descrito na secção de desenho, o sistema MixCloud implementa um serviço de armazenamento de dados replicado e distribuído orientado para a *cloud*.

O MixCloud utiliza vários repositórios de dados em simultâneo para replicação dos dados. Desta forma procura fornecer elevada disponibilidade e performance, explorando as melhores características de cada sistema. Um dos desafios é garantir que o estado final dos dados armazenados nos repositórios converge, com o mínimo de comunicação entre os mesmos. Pretende-se igualmente, implementar um serviço que suporte uma interface genérica que abstraia as aplicações dos clientes dos repositórios subjacentes, eliminando a dependência entre estes.

O MixCloud é constituído pelos componentes *ClientProxy* e *ServerProxy* que implementam o serviço. O primeiro estabelece a conexão entre os cliente e os seus dados e o segundo as garantias de serviço e gere o estado das réplicas armazenadas no repositório a que está associado.

Neste capítulo apresenta-se a implementação do protótipo do sistema, o qual foi realizado em Java, focando os aspectos essenciais e não triviais. Na primeira parte é detalhada a biblioteca de comunicação implementada, segue-se o *middleware* do lado do cliente e depois os servidores do lado dos repositórios de dados.

4.1 Biblioteca *Multicast*

No decorrer do desenvolvimento do cliente MixCloud foi criada uma biblioteca *multicast*, específica ao contexto, para suporte das comunicações entre o cliente e múltiplos servidores remotos. Pretende-se implementar uma biblioteca simples que ofereça garantias FIFO das operações do cliente no envio para os servidores. As primitivas de comunicação que

oferece dividem-se em *unicast*, para o envio apenas a um servidor, e *multicast*, para envio a um conjunto de servidores.

Como apresentado na figura 4.1, a biblioteca *multicast* é composta pelos componentes principais *MSocket*, *TSocket* e *TransportSocket*. O componente *MSocket* funciona como ponto de entrada das operações do cliente, implementando as primitivas de comunicação. Adicionalmente é responsável por iniciar os restantes componentes. O *TSocket* estabelece um canal de comunicação TCP ponto-a-ponto com um servidor. No *TransportSocket* está implementado o protocolo de comunicação que utiliza o *TSocket* para enviar e receber informação transmitida entre o cliente e um servidor.

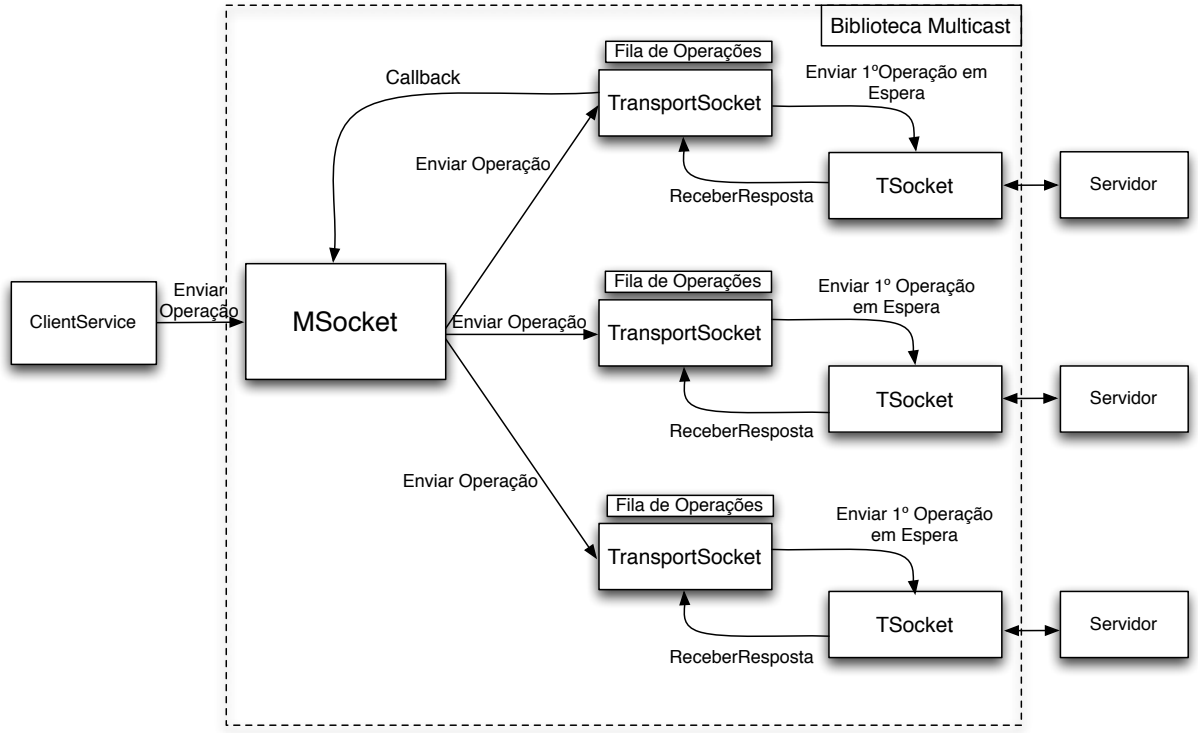

Figura 4.1: Desenho interno da Biblioteca *Multicast*.

Todos as operações são enviado através do componente *MSocket* utilizando apenas um dos dois métodos: $sendOneWayMessage(\textit{operação}, \textit{endereço\_servidor})$, para envio da operação apenas para o servidor indicado no endereço; $sendMessage(\textit{operação})$, que envia a operação para o conjunto de servidores conhecidos.

O *TSocket* disponibiliza as funcionalidades base para criar, abrir e fechar um canal de comunicação realizado por um *socket TCP* e dois específicos para o envio e recepção de operações/respostas. O envio de uma operação é feito através do método $sendMessage(\textit{operação})$ e a recepção através do método $receiveMessage()$, que espera sincronamente por uma resposta do servidor e a entrega ao *TransportThread*.

Por fim, o *TransportThread* implementa internamente uma fila de operações que garante a ordem FIFO da seguinte forma: a inserção de novas operações é efectuada à

"cauda"da fila, após a última operação; a próxima operação a enviar é retirada à "cabeça"da fila, correspondendo à operação mais antiga mantida na fila. Adicionalmente, todas as operações na fila que contenham um relógio inferior a um limite máximo indicado podem ser removidas, com o objectivo de eliminar operações antigas. A inserção de novas operações é feita através do método $insertMsg(\textit{operação})$, a remoção da operação mais antiga através do método $removeMsg()$ e a remoção de várias operações pelo método $deleteOldOperations(limite)$. O método de remoção de uma operação é bloqueante, isto é, bloqueia o processo se não existir operações para remover, saindo deste estado após ser inserida uma nova operação. Deste modo permite ao componente poupar recursos enquanto espera por operações vindas do cliente.

Algoritmo 4.1 Implementação do *TransportThread*

```
Inicialização.
socket ← TSocket(endereço do servidor)
fila ← OperationQueue()
id ← identificador do cliente

Thread de Trabalho.
while aplicação_activa do
  operação ← fila.removeMsg()
  socket.sendMsg(operação)
  resposta ← socket.receiveMsg()
  vector_versão ← resposta.getVersionVector()
  updateTimeVector(vector_versão)
  entregarResposta(resposta)
  fila.deleteOldOperations(vector_versão[id])
end while
```

É de notar que existe uma fila por cada servidor e essas filas são geridas de forma independente. Isto permite ao cliente adaptar-se aos diferentes ritmos de trabalho de cada servidor, garantindo que todos as suas operações chegam aos diferentes servidores pela mesma ordem, mesmo que em momentos diferentes. Igualmente, se algum dos servidores ficar incontactável, os restantes podem prosseguir sem qualquer tipo de impacto no cliente. As falhas de comunicação que ocorram são assim invisíveis a utilizador e podem ser resolvidas em *background*.

O protocolo de comunicação é executado ciclicamente até que o processo da aplicação do cliente termine o trabalho. Seguindo o algoritmo 4.1, o processo começa por pedir a operação mais antiga da fila de operações. Em seguida a operação é enviado para o servidor através do componente *TSocket* que espera sincronamente pela resposta. Ao receber uma nova resposta, entrega-a ao *TransportThread* que, por sua vez, actualiza o vector versão do cliente com o do servidor vindo na resposta, como descrito na secção 3.3. Apenas a primeira resposta recebida é entregue ao componente *service* via *callback* registado por si, sendo as restantes descartadas. Por questões de optimização da fila de operações, são eliminadas da fila as operações já aplicadas pelo servidor que foram transmitidas pela

via de sincronização. O limite é dado pelo valor registado na posição do cliente no vector versão vindo do servidor.

A gestão dos canais de comunicação está a cargo da biblioteca *multicast*, mantendo durante a sessão os canais sempre activos. Em caso de falha de algum destes, tenta re-estabelecer a conexão com o servidor automaticamente. Este tipo de falhas devem ser mascarados do serviço de forma a que este continue a funcionar mesmo que algum dos servidores deixe de estar disponível.

No lado do servidor encontra-se um módulo de comunicação específico de suporte à biblioteca *multicast*. Neste módulo está implementado o protocolo de comunicação que descreve como um novo canal de comunicação é estabelecido entre o cliente e servidor e como as operações devem ser recebidas e as respostas devolvidas. Como neste contexto são utilizados *sockets TCP*, o servidor utiliza um *server socket* para receber novas operações de contacto e criar o canal de comunicação.

4.2 Cliente MixCloud

A ligação da aplicação ao serviço MixCloud é efectuada através do componente *Client-Proxy* que funciona na biblioteca, acoplada à aplicação, no qual é disponibilizada a API genérica do serviço. Internamente, o *ClientProxy* implementa a parte do serviço que faz a gestão de envio de operações e recepção de respostas. Esta estratégia permite esconder das aplicações a estrutura do sistema e do serviço em termos dos repositórios em uso e da localização dos dados.

A ligação do cliente para o resto do sistema segue o modelo $1 - N$ em que cada operação é enviada através da API do serviço no cliente e difundida pelos N servidores disponíveis no momento. No *ClientProxy* foi implementada a biblioteca *multicast* que possibilita estabelecer canais de comunicação TCP ponto-a-ponto entre o cliente e cada servidor. A gestão destas ligações está a seu cargo, devendo manter as ligações sempre activas e em caso de falha de uma ligação garantir que o serviço não seja afectado.

Mapeamento das Operações

As operações implementadas na MixCloud API são mapeadas internamente em objectos no sistema com que o serviço trabalha para transmitir a informação internamente para o servidor.

Propõem-se uma organização hierárquica destes objectos como apresentado na figura 4.2, de forma a tornar simples a estruturação da informação, poupando assim recursos. No topo da hierarquia encontra-se o elemento *Operação* que representa qualquer operação sem discrição. No nível abaixo vem a *Operação do Cliente* que contém todos os campos de dados comuns a uma operação, como a estampilha temporal, o vector versão e o número de sequência da operação. No último nível estão as operações que contêm os campos correspondentes aos argumentos das operações da API. Por exemplo, um objecto

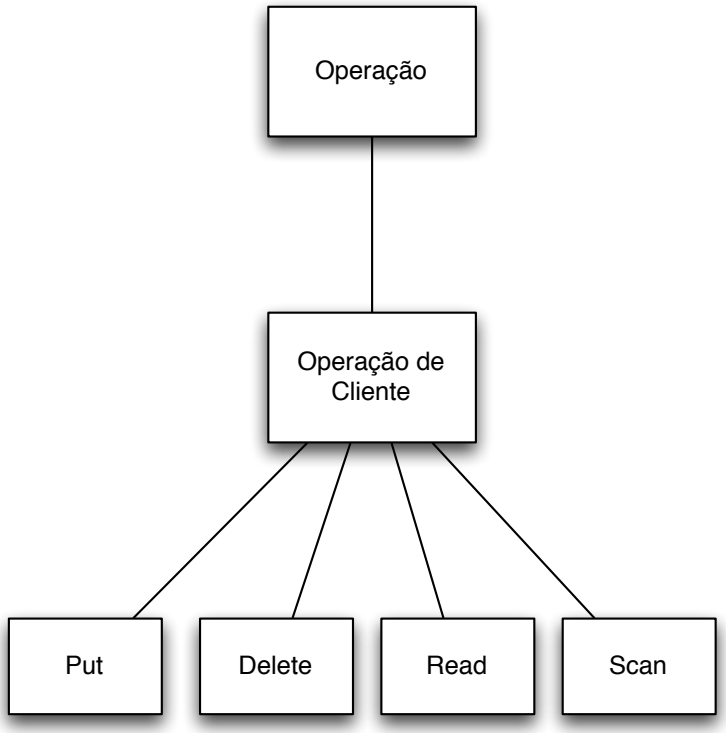

Figura 4.2: Hierarquia de Operações.

Put é constituído pelos campos família, chave, e lista de atributos e valores a inserir/actualizar. Estas operações têm acesso aos campos das operações dos níveis acima.

As operações implementam mecanismos de serialização e desserialização para poder enviar/receber os seus dados para o/do servidor. O mecanismo de serialização converte todos os campos da operação juntamente com os campos da *Operação do Cliente* em conjuntos de bytes e envia-os pelo canal de comunicação para o servidor. Por outro lado, o mecanismo de desserialização lê do canal os conjuntos de bytes vindos do servidor e converte-os para as representações correctas dos campos da operação.

A cada tipo de operação é atribuído um valor inteiro com o intuito de facilitar a sua distinção, sendo que o *Put* tem o valor 1, o *Delete* 2, o *Read* 3 e o *Scan* 4.

Serviço

A gestão do serviço MixCloud no *ClientProxy* encontra-se centralizado no componente *service*. Concretamente, este implementa o protocolo de execução de uma operação, desde que é emitida pelo cliente, passando pelo envio para os servidores, até receber resposta à operação, se for o caso. Só após a execução de uma operação terminar recomeça o processo para uma próxima operação. A utilização de componentes adicionais, como o *time*, permite manter o serviço coerente e correcto.

O componente auxiliar *time* gere o estado do relógio lógico e vector versão conhecidos

Algoritmo 4.2 Bloco de Inicialização do *Serviço*

Inicialização.
$clientId \leftarrow$ identificador do cliente
número_sequência $\leftarrow 0$
$servidor \leftarrow$ endereço IP do servidor que irá responder às operações de Leitura
$time \leftarrow$ nova instância de *time*
$comm \leftarrow$ nova instância de *comm*

pelo cliente. As primitivas disponibilizadas por este componente para ler/actualizar o relógio lógico são $getTimestamp()$ e, para ler/actualizar o vector versão, $updateTimestamp(valor)$ e $getTimeVector()$ e $updateTimeVector(vector\_versão)$ para ler/actualizar o vector versão. A actualização do vector versão do cliente consiste na fusão deste com o que vem do servidor dado em argumento. Uma fusão corresponde à união entre dois vectores, $V_{cliente}$ e $V_{servidor}$, quando para cada posição i, de ambos, $V_{cliente}[i] = Max(V_{cliente}[i], V_{servidor}[i])$ (o método $Max(valor_1, valor_2)$ retorna o máximo dos valores em argumento). Assume-se que tanto o cliente como o servidor contêm vectores versão da mesma dimensão, isto é, conhecem todo o conjunto de clientes do sistema. Após actualização do vector versão, o valor do relógio lógico do cliente é alterado para o valor máximo contido no vector versão recebido, caso algum seja maior que o actual valor do relógio.

A comunicação entre o componente *service* e a biblioteca *multicast* é realizada indirectamente por meio de um componente *comm*. Este componente oferece duas primitivas simples de envio de operações: *sendOneWay* e *send*, que correspondem, respectivamente, às primitivas *sendOneWayMessage* e *sendMessage* da biblioteca *multicast*. Esta aproximação permite esconder do *service* qual a biblioteca em uso e adicionar um mecanismo de suporte de adição rápida de novas bibliotecas com baixo nível de modificação.

No cliente implementamos um mecanismo baseado em *callbacks* para suportar a recepção de respostas sincronamente no componente *service*. Um *callback* funciona como um pedido de notificação de resposta para uma operação enviada. O processo que cria o *callback* bloqueia a sua execução até que uma notificação seja enviada a indicar a recepção da resposta. Este mecanismo é implementado através das primitivas $get()$ e $response(id\_da\_operação)$ implementadas no objecto de *callback*: o get, quando invocado, bloqueia o processo até receber uma resposta; o $response$ permite entregar apenas uma nova resposta. O mecanismo funciona da seguinte forma: o componente *service*, antes de enviar a operação, cria um novo *callback*, contendo o número de sequência da operação, regista-o e invoca o método get. No lado da biblioteca, ao receber uma resposta, verifica se existe algum registo para a operação correspondente e, caso positivo, invoca o método $response$ e entrega a resposta. Para o efeito foi criado um mapa no componente *comm* que regista os *callbacks* por número de sequência da operação a que correspondem.

No contexto do MixCloud foi introduzido um novo campo nas operações que indica o nível de consistência que o cliente pretende para execução da operação. Actualmente são apenas suportados os níveis *Zero* e *One*: no primeiro a operação é executada assincronamente em todos os servidores sem que o cliente espere pela resposta de algum e no

Algoritmo 4.3 Implementação das operações de Leitura no Serviço

Operação de Leitura - Read(argumentos) e Scan(argumentos).

número_sequência $\leftarrow$ *número_sequência* $+ 1$
relógio $\leftarrow time.getTimeStamp()$
vector_versão $\leftarrow time.getTimeVector()$
operação $\leftarrow \{Read/Scan\}(identificador,$ *número_sequência*, *relógio*, *vector_versão*, $argumentos)$

if *nível_consistência* $= One$ **then**
  $callback \leftarrow OneResponseHandler()$
  registar callback juntamente com o número de sequência da *operação*.
  $comm.sendOneWay($*operação*$, servidor)$
  $resposta \leftarrow callback.get()$
  retornar resposta
end if

Algoritmo 4.4 Implementação das operações de Escrita no Serviço

Operação de Escrita - Put(argumentos) e Delete(argumentos).

número_sequência $\leftarrow$ *número_sequência* $+ 1$
relógio $\leftarrow time.getTimeStamp()$
vector_versão $\leftarrow time.getTimeVector()$
operação $\leftarrow \{Put/Delete\}(identificador,$ *número_sequência*, *relógio*, *vector_versão*, $argumentos)$

if *nível_consistência* $= Zero$ **then**
  $comm.send($*operação*$)$
  retornar resposta genérica que indica sucesso
end if
if *nível_consistência* $= One$ **then**
  $callback \leftarrow OneResponseHandler()$
  registar callback juntamente com o número de sequência da operação
  $comm.send($*operação*$)$
  $resposta \leftarrow callback.get()$
  retornar resposta
end if

segundo o cliente espera sincronamente pela resposta do servidor mais rápido. As operações *Read* e *Scan* suportam unicamente o nível de consistência *One* porque é necessário esperar pelo menos por uma resposta.

O algoritmo 4.2 corresponde ao bloco de inicialização do serviço MixCloud. O conjunto de atributos que compõe o serviço é formado por o número de identificação do cliente, o número de sequência atribuído à ultima operação, o endereço do servidor a responder às operações de leitura e uma referência ao componente *Time* e ao *comm*. Nos algoritmos 4.3 e 4.4 estão apresentados os esquemas gerais de execução de uma operação de leitura e escrita no cliente, respectivamente. Estes algoritmos possuem um bloco inicial comum onde é calculado o novo número de sequência e uma estampilha temporal,

obtido o vector versão actualizado e criado um novo objecto que representa a operação a executar. Nesta operação são introduzidos os argumentos de entrada, dados pelo utilizador, juntamente com os dados obtidos no passo anterior.

Em seguida, consoante o nível de consistência indicado, calcula-se qual o número de respostas a esperar para aceitar e terminar a execução de uma operação. Se o nível for *One*, cria-se e regista-se um novo callback utilizando o número de sequência da operação, envia-se a operação através da primitiva $send$ para todos os servidores e invoca-se a primitiva $get()$ que bloqueia o processo que executa o serviço. Ao receber uma resposta, o processo retoma a execução e a resposta é entregue directamente à aplicação. No caso do nível *Zero*, a operação é enviada usando a primitiva $send$ e o serviço retorna à aplicação uma resposta genérica de sucesso.

4.3 Servidor MixCloud

O servidor MixCloud é composto por um conjunto de *ServerProxies* que estabelecem a comunicação entre o cliente e os repositórios. A sua acção passa por receber as operações vindas dos clientes, reencaminhá-las para o repositório associado que trata de aplicar as acções solicitadas e devolver resposta ao cliente. Os repositórios individualmente não são capazes de manter o nível de coerência desejada para o serviço MixCloud, pois implementam exclusivamente mecanismos de consistência interna. A solução passa por implementar no componente *ServerProxy* as garantias de consistência do serviço utilizando um mecanismo de transformação de operações que permite a que os repositórios, aplicando o mesmo conjunto de operações mas por ordens diferentes, convirjam para um estado final consistente.

Neste contexto, foi adoptado um mecanismo de sincronização com o intuito de acelerar o processo de convergência de estado entre servidores. Os servidores trocam os vectores de versão entre si para verificar se algum se encontra dessincronizado. Comparado as posições dos vectores versão, local e recebido, verifica-se que um servidor encontra-se dessincronizado se numa mesma posição o vector recebido possui um valor superior ao valor no vector local. Em seguida pede-se as operações em falta. De outra forma a sincronização pode ser invocada se o vector local do servidor estiver atrasado relativamente ao vector recebido através de uma operação enviada por um cliente.

Os servidores comunicam com os repositórios através da biblioteca específica do respectivo repositório. O protótipo actual suporta três repositórios chave/valor: Cassandra [LM10], HBase [HBa11] e MongoDB [Mon11].

Tempo

O tempo reveste-se de uma elevada importância no estabelecimento das garantias do serviço MixCloud, em particular na manutenção da consistência no estado dos servidores e das garantias na execução de cada tipo de operação emitida pelo cliente. No servidor

é armazenada informação sobre o estado local e global do sistema. O estado local é representado por um vector versão que regista as operações já vistas de todos os clientes conhecidos pelo servidor. O estado global resulta da agregação dos vectores versão contidos nas operações vindas dos clientes e representa o estado conjunto dos vectores locais de cada servidor. Este último permite verificar se um servidor se encontra sincronizado com os restantes a cada momento e assim acelerar o processo de sincronização no caso negativo.

Mantém-se o registo dos vectores versão locais de cada servidor trocados durante a fase de sincronização. A utilização destes registos tem a finalidade de possibilitar o cálculo do ponto de sincronização entre os servidores após cada sincronização. O ponto de sincronização é representado por um vector versão cujo o valor de cada posição é o valor mínimo da posição respectiva dos vectores versão dos servidores. Definido o ponto de sincronização, pode-se descartar todas as operações do estado de um servidor que tenham sido geradas previamente a este ponto, isto é, que possuam um relógio inferior ao valor no vector versão contido na posição do cliente respectivo. Este passo é importante para eliminar operações sem utilidade e libertar espaço em memória usado pelo *log*, permitindo optimizar as pesquisas no *log*.

Comunicação

Os servidores implementam módulos de comunicação genéricos para que possam ser adaptados ou substituídos conforme as necessidades exigidas ao serviço MixCloud. A integração de novos módulos de comunicação faz-se de forma simples e directa, sendo a comunicação com os restantes componentes realizada por meio de uma classe abstracta que esconde a pilha de comunicação. Obtém-se um nível de independência elevado entre componentes, cujas alterações no protocolo de comunicação não são visíveis nem produzem efeitos colaterais nos demais componentes. Esta aproximação permite que um programador comece a trabalhar sobre os módulos de comunicação mesmo sem saber qual a estrutura e pormenores de implementação no servidor.

O componente externo de comunicação que permite os clientes comunicarem com os servidores está no protótipo actual implementado sobre a classe *SimpleTCPComm*, que estende as funcionalidades da classe *IServerComm*. A cada novo canal estabelecido com o cliente, o servidor cria uma *thread* que fica responsável por gerir o fluxo de informação trocado com o cliente, durante o período em que esse canal se mantém activo. Na presença de um nova operação, a *thread* entrega-a ao componente *MessageDispatcher* e espera pela resposta a enviar ao cliente.

Ao nível interno, o protocolo de comunicação entre servidores encontra-se especificado na classe *IComm*, mais concretamente na classe *InternalTCPComm*. As mensagens de serviço são distinguíveis por identificadores únicos. Apenas um pedido é tratado de cada vez, com o objectivo de garantir consistência nas respostas. Actualmente suporta apenas pedidos de sincronização no qual, em conjunto com o componente *synchronize* e

messageDispatcher, sincroniza o estado de um outro servidor com o seu e vice-versa.

Message Dispatcher

O componente *Message Dispatcher* funciona como um *proxy* interno que liga os restantes componentes do servidor entre si, a fim de tornar o serviço MixCloud funcional. Este componente apresenta uma estrutura simples contendo dois métodos para o tratamento de operações vindas de um cliente e de um servidor. Adicionalmente, implementa uma fila de operações dos clientes que ainda não foram tratadas. A utilização desta fila permite que o servidor consiga definir uma ordem pelo qual serão executadas as operações. Por outro lado, os pedidos de sincronização invocados pelos servidores são imediatamente tratados sem ordenação prévia com a fila de operações. Tal deve-se ao facto de o processo de sincronização ter de ocorrer no menor espaço de tempo possível, de forma a não causar um impacto significativo nos tempos de resposta ao tratamento das operações dos clientes.

Com o objectivo de aumentar o grau de concorrência dentro do servidor, são utilizadas várias *threads* trabalhadoras, independentes entre si, que continuamente retiram uma operação da fila e encaminham-a para o serviço, a fim de ser executada. Deste conjunto, apenas uma *thread* é atribuída exclusivamente para o tratamento de pedidos de sincronização. O conjunto das *threads* pode variar em número consoante o escolhido pelo administrador de sistemas tendo em conta o número esperado de clientes e as capacidades da máquina onde corre o servidor.

O módulo de comunicação e os trabalhadores não comunicam directamente entre si, pois pertencem a *threads* de execução diferentes. Desse modo, adoptou-se uma solução baseada em *callbacks*, como apresentada no cliente na secção 4.2. Para tal, o *Message Dispatcher* mantém um mapa no qual armazena os identificadores dos *callbacks* registados pelo módulo de comunicação externo.

O protocolo de execução de uma operação vinda de um cliente está descrito no algoritmo 4.5. O primeiro passo consiste em actualizar o vector versão global do servidor com

Algoritmo 4.5 Protocolo de Execução das Operações

$id \leftarrow identificador\ da\ operação$
$vector\_cliente \leftarrow vector\ versão\ do\ cliente\ vindo\ na\ operação$
actualizar vector versão global no componente tempo com o $vector\_cliente$
verificar se o servidor encontra-se actualizado
if $id = 1 \vee id = 2$ **then**
  entregar a operação ao serviço para transformação
  enviar operação transformada ao repositório
  actualizar vector versão local na posição correspondente ao cliente dono da operação
else if $id = 3 \vee id = 4$ **then**
  enviar operação ao repositório
end if
registar resposta no mapa de *callbacks*

o da operação do cliente e verificar se o servidor se encontra actualizado. Se a resposta for negativa, inicia-se a fase de sincronização para garantir que a operação executa sobre um estado consistente. Consoante o tipo de operação, seguem caminhos diferentes de execução. Depois de receber a resposta, adiciona-lhe o vector versão local do servidor e regista-a no *callback* respectivo.

Sincronização

A sincronização consiste num processo de troca de informação entre os servidores onde são trocados os vectores versão que reflectem o estado de cada servidor, permitindo verificar se algum se encontra dessincronizado. A sincronização também é invocada se o vector local do servidor estiver atrasado relativamente ao vector recebido numa operação dum cliente.

A fase de sincronização realiza-se periodicamente, sendo o período estaticamente definido antes do sistema executar. A cada novo período o protocolo de sincronização é activado e escolhe-se um servidor, de entre a lista dos conhecidos, para estabelecer contacto. Em paralelo pode receber pedidos de sincronização. Apenas um dos processos pode executar num determinado momento, não sendo possível sincronizar em paralelo com dois servidores. De forma a evitar eventuais *deadlocks* nos servidores, os pedidos enviados têm associado um tempo de espera para serem atendidos. Após o tempo terminar é lançada uma excepção e o servidor que emitiu o pedido espera até ao próximo ciclo para efectuar a sincronização. Consoante o desenrolar do processo de sincronização o período pode variar, ou seja, se o protocolo de sincronização foi correctamente executado o período aumenta para o dobro; caso contrário, ou na ocorrência de excepções, o período volta ao valor inicial. Esta pequena optimização permite distribuir o trabalho de sincronização pelo tempo e pelos servidores, não sendo a sincronização activada sempre em intervalos de tempo pequenos, evitando um impacto maior no tempo de execução das operações dos clientes.

Algoritmo 4.6 Protocolo de Sincronização

$servidor \leftarrow$ *endereço de um dos servidores da lista de servidores activos*
enviar identificador do servidor e vector versão para o endereço $servidor$
$< id, vector > \leftarrow identificador\ e\ vector$ *versão do outro servidor*
pedido_operações $\leftarrow$ *lista de pedido de operações que* $resulta$ *da comparação entre os* $vectores$ *versão*

enviar *pedido_operações* para endereço $servidor$
lista_operações_remoto $\leftarrow receber\ lista$ *de operações de* $servidor$
enviar *lista_operações_remoto* para o componente *Message Dispatcher*

Em paralelo.
$pedido\_remoto \leftarrow lista$ *de pedido de operações* $enviado\ pelo\ outro\ servidor$
lista_operações $\leftarrow obter\ lista$ *de operações a* $enviar$
enviar *lista_operações* para endereço $servidor$

No algoritmo 4.6 apresenta-se, em pseudo-código, o protocolo de sincronização, descrito em seguida. Quando o protocolo é activado, verifica se existe algum processo de sincronização em marcha. No caso negativo, escolhe-se um servidor e envia-se-lhe um pedido de sincronização com o identificador e o vector versão local. Em retorno recebe o identificador e o vector versão do outro servidor e prossegue com o registo do vector. Neste ponto é realizada a verificação dos vectores versão seguindo o método descrito na secção 3.5. Deste processo é gerada a lista das operações em falta no estado do servidor, devidamente identificada por cliente, e enviada para outro servidor. Ao receber a lista de operações pedida, entrega ao componente *Message Dispatcher* a fim de serem aplicadas no seu estado. Paralelamente recebe o pedido das operações em falta do outro servidor.

O protocolo de sincronização e a sua periodicidade é controlada por uma *thread* dedicada a este processo, que executa o protocolo descrito acima e actualiza a periodicidade consoante o seu término. No entanto, o período de espera pode ser interrompido se existir evidências que o estado encontra-se desactualizado. Tal é conseguido através da comparação entre o vector versão local do servidor e o vector versão contido na operação de um cliente. Dependendo da operação em causa são tomadas acções diferentes: se for uma escrita, inicia-se a fase de sincronização apenas se a diferença entre estados for significativa; no caso de uma leitura, a fase de sincronização é iniciada imediatamente se as escritas das quais esta depende ainda não se encontram aplicadas no estado do servidor. O primeiro caso funciona como um auxiliar à fase de sincronização acelerando o processo se o tempo de espera for elevado e no segundo caso garante-se que as leituras lêem sempre um estado actualizado consoante as acções anteriormente efectuadas pelo cliente.

Serviço

O serviço MixCloud divide-se em duas áreas principais: o cliente implementa a API de acesso ao sistema, enquanto que o servidor,assume o papel principal na manutenção da consistência do estado dos repositórios a cada nova operação executada. Optou-se por uma solução centralizada, utilizando apenas um componente para implementar o serviço, com objectivo de facilitar e agilizar o processo de implementação, manutenção e correcção de erros.

No serviço é mantido o registo, ou *log*, de todas as operações de escrita executas correctamente, ordenadas pela sua estampilha. A implementação do *log* está realizada sobre duas estruturas diferentes consoante o mecanismo a suportar por estes: transformação de operações ou sincronização de servidores. Numa primeira aproximação as operações são organizadas num mapa por chave de acesso, composta pelos elementos família e linha, e o valor é o conjunto de operações que produziram alterações nos atributos sobre essa chave. Esse mapa é denominado por *StructuredLog*. O algoritmo de ordenação e transformação de operações utiliza esta estrutura para obter a lista de operações que acederam à mesma chave de uma operação a executar. Um segundo *log*, *ClientLog*,

está também implementado num mapa cuja chave é o identificador do cliente associado à lista de operações emitida por esse cliente. A diferente organização permite tornar a pesquisa de operações por cliente mais eficiente, possibilitando diminuir o tempo de sincronização entre servidores. No processo de sincronização faz-se a pesquisa no servidor actualizado para obter a lista de operações em falta por cliente. Ambos os *logs* suportam os métodos $put(\text{operação})$, para adicionar uma nova operação transformada, e o $deleteOldOperations(estampilha)$, onde é indicado uma estampilha em argumento e todas as operações que se encontrem abaixo dessa estampilha são eliminadas. A inserção e a remoção de operações são sempre realizadas em ambos os *logs* para manter as suas consistências. Apagamos operações antigas dos *logs* com intuito de libertar memória e tornar as pesquisas nas estruturas eficientes.

As operações inseridas nos *logs* apresentam diferenças relativamente às operações do cliente, de onde foram retirados os campos número de sequência e o vector versão e adicionado um campo contendo o conjunto dos atributos acedidos. Apenas na operação *Put* foi criado mais um campo que indica o conjunto dos valores a adicionar/alterar, devidamente relacionados com os atributos. Este campo tem a particularidade de ser dinâmico e adaptar-se automaticamente consoante se pretende que os valores sejam armazenados localmente ou acedidos remotamente, através de uma pesquisa no repositório utilizando o conjunto de atributos, quando precisos. Armazenar localmente os valores permite efectuar pesquisas mais rápidas mas ocupa espaço em memória e pode gerar um grande *overhead*. No acesso remoto verifica-se o contrário. As operações possuem um método $deleteRemoteFields(lista\_de\_atributos)$ que à lista de atributos dada em argumento remove os elementos em comum da lista de atributos da operação.

Algoritmo 4.7 Protocolo de Ordenação e Transformação de Operações

$log\_op \leftarrow$ *operação* $traduzida\ para\ log$
$chave \leftarrow log\_op.chave()$
if $chave$ *não* $existe$ **then**
  inserir chave e nova lista de operações contendo apenas a log_op
  retornar log_op
end if
$pos \leftarrow getPosition(log\_op)$
$lista \leftarrow$ *obter a lista de operações que acedem a mesma chave de* log_op *e se encontrem posicionadas após a posição* pos.
$transformada \leftarrow$ *transformação*$(log\_op, lista)$
$log.put(transformada)$
retornar $transformada$

O algoritmo de ordenação e transformação, descrito em 4.7 começa por converter a operação do cliente recebida em argumento para uma operação do *log*. Em seguida é verificado se já foi criada no *log* uma entrada para a chave correspondente à chave de acesso da operação. Este passo tem de ser atómico por *thread* para evitar criar duas entradas de uma mesma chave. No caso de não existir uma entrada, é adicionada uma nova

chave e, juntamente com esta, insere-se a lista de operações, contendo apenas a operação inserir que não sofre transformações. Caso contrário, inicia-se a fase de ordenação seguida da fase de transformação. O método $getPosition(\text{operação})$ é utilizado pela fase de ordenação para obter a posição da operação no log. O conjunto das duas fases tem de ser executado atomicamente por chave de acesso, isto é, apenas uma operação pode ser executada por chave num dado momento. Assim garante-se que não existem alterações concorrentes sobre os mesmos dados, o que pode causar disrupção no estado. Após esta fase é chamado o método $put(\text{operação})$ de ambos os *logs* que insere uma operação transformada, respeitando a ordem natural, e em seguida é executada no repositório.

Na fase de sincronização é utilizado o método $getOperations(\textit{identificador\_cliente}, \textit{relógio})$ para obter as operações de um dado cliente, referenciado pelo seu identificador, que possuam um relógio igual ou superior ao indicado no segundo argumento. A pesquisa é feita de forma simples: o identificador dá acesso à lista de operações, itera sobre a lista à procura das operações que respeitem a condição indicada e retorna a lista de operações encontrada.

4.4 Repositórios de Dados

Os repositórios de dados utilizados neste sistema apresentam modelos de dados diferentes entre si. O Cassandra e o HBase apresentam um modelo semelhante ao nosso, isto é, orientado à família, linha e coluna, apresentado inicialmente pelo sistema BigTable [CDG+08]. Uma família contém um conjunto disjunto de colunas, que guardam os dados da aplicação, e um número arbitrário de linhas, onde a cada linha está associada um conjunto de colunas. No MongoDB, o modelo é orientado ao documento, contendo uma chave única ordenada no sistema. O conteúdo destes documentos está organizado por campos associados a valores. Grupos de documentos são organizados em colecções que podem ser vistas como tabelas tradicionais.

A adaptação do nosso modelo de dados aos modelos do Cassandra e HBase é directa. O modelo do MongoDB é facilmente adaptável, ao relacionar os documentos às linhas, os campos dos documentos às colunas e as colecções correspondem às famílias. Assim é possível, de forma transparente para a aplicação, comunicar com os repositórios através de uma biblioteca que implementa este modelo de dados simples, independente da plataforma subjacente.

5

Resultados

Para a avaliação do MixCloud, usou-se o *YCSB Benchmark* [CST+10] que permite medir o desempenho de repositórios chave/valor. Os resultados obtidos são o desempenho do repositório em número de operações por unidade de tempo (*throughput* em *ops/s*) e as latências médias da execução dos conjuntos de operações de leitura e escrita, em milisegundos. Os testes são definidos por meio de um *workload* onde se indica o número de operações a executar e dessas, qual a percentagem de leituras e de escritas.

Os workloads representam diferentes tipos de aplicações que poderiam ser implementadas sobre o sistema MixCloud, desde aplicações para guardar o estado da sessão, ou apenas de leitura de dados, até acesso a bases de dados que mantêm grandes volumes de dados. Os testes correspondem a uma carga de 200.000 operações. A distribuição de chaves nos testes é não uniforme para garantir que as chaves seleccionadas não sejam sequenciais. Cada operação de escrita apenas altera o valor de um atributo escolhido aleatoriamente, associada a uma família e chave. As leituras devolvem os valores de todos os atributos de uma linha. A família é um valor estático igual para todas as operações, enquanto as chaves das linhas são escolhidas aleatoriamente. Assim, garante-se uma distribuição das operações coerente e justa pelo espaço de chaves do sistema. As linhas são formadas por cinco atributos que armazenam uma palavra gerada aleatoriamente com cinquenta caracteres. O tamanho aproximado das linhas é de 250 bytes.

O ambiente de teste é constituído por um cluster de 4 nós, com servidores Dell, processadores AMD Opteron Quad-core 2376 a 2.3Ghz, 16 Gb de memória, a executar o sistema operativo Linux Debian 5. As máquinas estão conectadas por uma rede Ethernet 1Gb. Três nós do cluster são usados para executar três servidores distintos e os respectivos repositório chave/valor. Nesta avaliação foram usados os repositórios HBase, MongoDB e Cassandra. O quarto nó do cluster executa a aplicação cliente.

Os resultados apresentados incluem: o acesso directo a cada um dos sistemas (Cassandra, HBase e MongoDB); o acesso ao sistema MixCloud configurado com apenas um servidor (MidCassandra, MidHBase e MidMongoDB), para ver o *overhead* do *middleware* sobre cada um dos repositórios; o acesso ao sistema MixCloud usando três repositórios chave/valor. Neste último ambiente, a sincronização do MixCloud está activada com periodicidade de 30 segundos. A execução dos testes tem duração média de cinco minutos. O primeiro gráfico de cada teste apresenta os resultados do desempenho geral e o segundo os resultados relativos à latência por operação.

5.1 Workload *heavy-write*

No primeiro ambiente de teste tenta-se simular aplicações com grandes volume de escritas, como por exemplo aplicações que armazenam o estado da sessão de trabalho. O workload respectivo é composto por 50% para leituras e as restantes escritas. As leituras são realizadas através de operações *Read* e as escritas *Puts*.

Os resultados apresentados nas figuras 5.1 e 5.2 permitem desde logo verificar diferenças que existem entre o vários repositórios de dados. Em termos de desempenho, o sistema MongoDB apresenta o melhor resultado. Tal é confirmado nos valores das latências, onde o tempo de execução de operações de leitura pelo MongoDB é menor comparativamente aos restantes. No entanto, ao nível da latência de execução de operações de escrita, o Cassandra obtém um melhor resultado, ainda que sem diferença considerável do MongoDB. Estas diferenças podem ser explicadas pelas implementações distintas de cada um dos sistemas: o Cassandra implementa um *log* sequencial de escritas e escreve as alterações em memória, enquanto o MongoDB implementa índices em memória para acesso eficiente aos dados em disco.

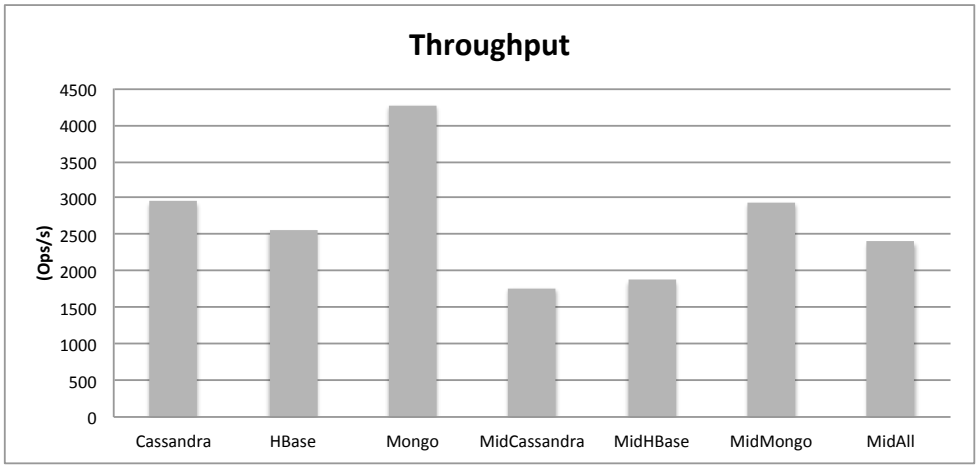

Figura 5.1: Desempenho do sistema (carga *heavy-write* com 50% leituras e 50% escritas).

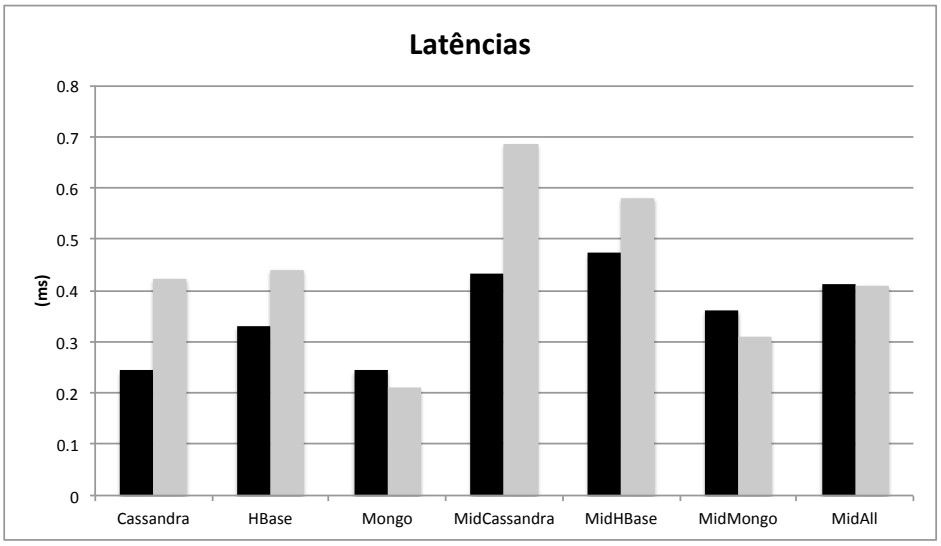

Figura 5.2: Latência das operações (carga *heavy-write* com 50% leituras e 50% escritas). As barras a preto representam a latência para as operações de escrita e as cinzentas para as de leitura.

Ao aceder aos sistemas através do sistema MixCloud, nota-se um decréscimo generalizado no desempenho, em média de 30%, e consequentemente um aumento na latência das operações. Este facto é devido ao *overhead* introduzido pelo sistema MixCloud, incluindo um passo adicional de comunicação face ao acesso directo ao servidor. No entanto, este decréscimo não é proporcional em todos os sistemas. Por exemplo, o MidCassandra apresenta uma diminuição de desempenho em cerca de 39% relativamente ao Cassandra, enquanto o MidHBase diminui o desempenho em cerca de 30% relativamente ao HBase. Uma justificação advém do facto dos sistemas serem diferentes entre si e apresentarem comportamentos distintos quando introduzido um novo *overhead* externo. Outra razão seria variações na carga dos CPUs e memória durante a execução do teste na máquina onde o repositório está alojado.

O MidMongo e o MidHBase obtêm um aumento de latência proporcional em ambas as operações em relação aos respectivos sistemas de base. Contrariamente, o MidCassandra apresenta uma subida superior para as leituras relativamente ao Cassandra. Na origem desta subida está possivelmente um *bottleneck* no MixCloud, resultando num maior tempo de execução para cada operação de leitura.

Os resultados do MixCloud configurado com os três repositórios permitem observar que não há melhoria no desempenho face à utilização de apenas um repositórios. Contudo, é melhorar o desempenho do serviço face aos sistema *middleware* com apenas um repositório, devido à distribuição de carga pelos vários repositórios para compensar o *overhead* introduzido.

Relativamente à latência, os resultados aproximam-se dos resultados obtidos pelo teste MidCassandra para as escritas e, pelo teste MidMongo para as leituras. O desempenho das escritas é superior, o que é explicado pelo facto do repositório com melhor

desempenho para as escritas não estar a executar leituras, tendo assim uma carga inferior à da experiência com apenas um repositório.

No entanto, pode-se verificar que o MixCloud consegue aproximar o seu desempenho aos sistemas sem o middleware ao agregar diversos repositórios. A adição de um repositório de alto desempenho nas escritas contribui para o aumento do desempenho do sistema, compensando assim o *overhead* introduzido pelas comunicações, transformação operacional e a sincronização.

5.2 Workload *read-mostly*

O *workload* que se segue é representativo de aplicações que realizam uma carga superior de leituras relativamente a escritas. Um exemplo deste tipo de aplicações pode ser um cliente para aceder a uma rede social, sobre o qual se realiza poucas actualizações ao estado do perfil do cliente, comparativamente ao número de leituras do estado dos perfis conhecidos armazenados no sistema. O teste consiste em realizar 95% de operações de leitura e 5% de escritas. As leituras são realizadas através da operação *Read* e as escritas *Put*. Os resultados estão contidos nos gráficos das figuras 5.3 e 5.4

Tal como no teste anterior, o sistema MongoDB é o melhor do teste, obtendo um desempenho duas vezes superior comparativamente aos seus homólogos. Em termos de latência, o MongoDB obtém valores semelhantes ao primeiro teste enquanto o Cassandra e o HBase pioram em 18% e 27%, respectivamente, o seu tempo de execução. Do conjunto, o MongoDB apresenta melhor desempenho na execução de operações de leitura, compensando o *overhead* das escritas. Em contraponto, o número pequeno de operações de escrita não compensa a latência causado pela leituras nos sistemas Cassandra e HBase, piorando significativamente o seu desempenho em relação ao teste anterior.

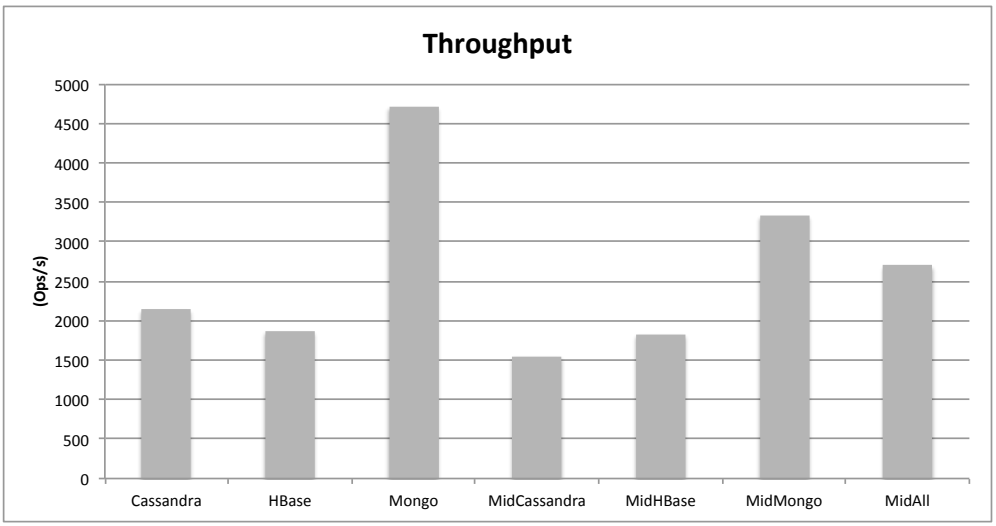

Figura 5.3: Desempenho do sistema (carga *read-mostly* com 95% leituras e 5% escritas).

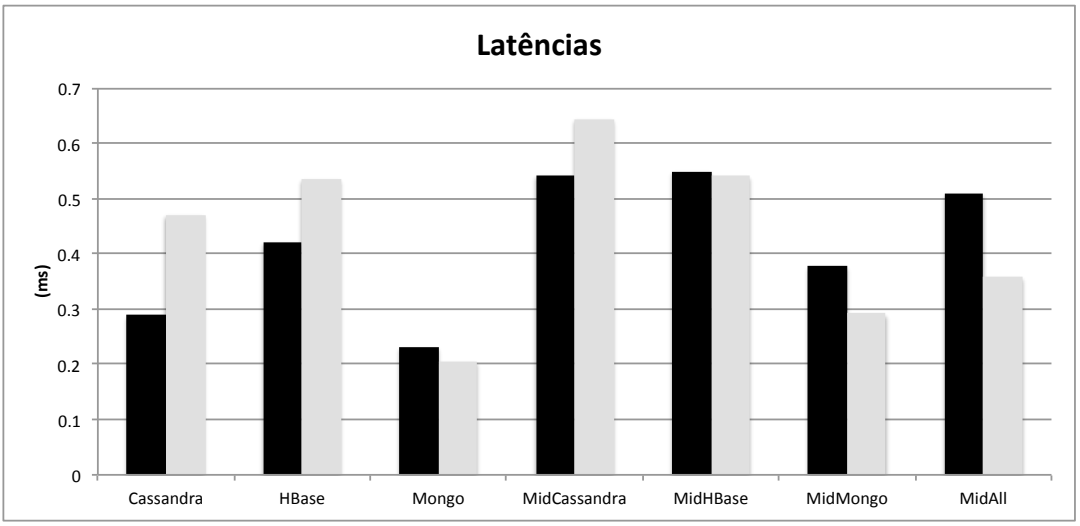

Figura 5.4: Latência das operações (carga *read-mostly* com 95% leituras e 5% escritas). As barras a preto representam a latência para as operações de escrita e as cinzentas para as de leitura.

Ao adicionar o *middleware* aos sistemas de base, verifica-se uma degradação do desempenho geral e um aumento na latência de execução de ambas as operações. Os sistemas MidCassandra e MidMongo aumentam a latência das operações em porporção com os valores obtidos pelos sistemas de base, Cassandra e Mongo, respectivamente. Por sua vez, o MidHBase apresenta apenas um aumento de latência das operações de escrita mantendo as de leitura.

A integração dos vários repositórios neste teste foi vantajosa, permitindo obter um desempenho superior ao sistema Cassandra e HBase. A latência das operações de leitura obtida pelo MixCloud aproxima-se do resultado do sistema MidMongo, enquanto nas escritas aproxima-se do MidCassandra. Podemos concluir que as leituras estão a ser efectivamente enviadas para o sistema mais rápido a executá-las, permitindo obter melhores desempenhos neste campo. Devido ao número reduzido de operações de escrita, o seu impacto é menor no desempenho MixCloud que tira maior proveito da execução das leituras. Observa-se nestes testes que efectivamente a agregação dos vários repositórios de dados permite melhorar o desempenho geral do sistema, verificando-se um desempenho que tende para o obtido quando usado directamente o MidMongo em termos de *throughput* e latências.

5.3 Workload *write-only*

O ambiente apresentado pelo workload *write-only* visa simular sistemas de registo de eventos que apenas produzam dados a armazenar para um determinado fim, como, por exemplo, popular uma base de dados com dados recolhidos numa rede de sensores. Assim o teste é composto apenas por operações de escrita (*Puts*).

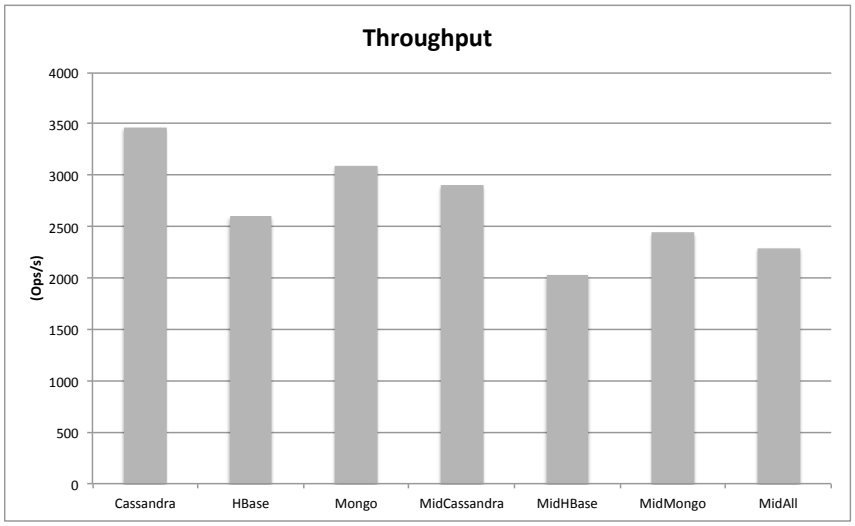

Figura 5.5: Desempenho do sistema (carga *write-only* com 0% leituras e 100% escritas).

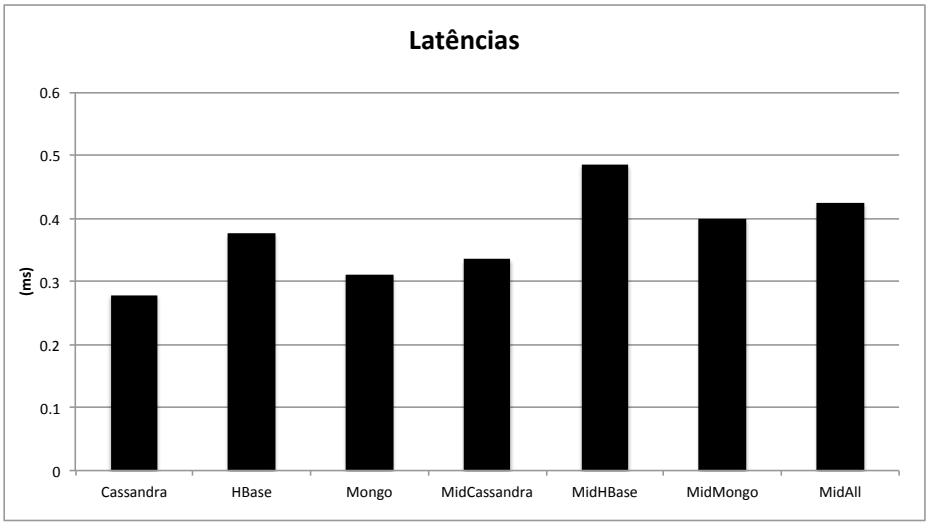

Figura 5.6: Latência das operações (carga *write-only* com 0% leituras e 100% escritas).

Ao contrário do que tem acontecido nos testes anteriores, o sistema Cassandra apresenta uma clara melhoria no desempenho, superando o HBase e o MongoDB. O desempenho das escritas no Cassandra é superior comparativamente aos testes anteriores, com aumentos aproximados de 17% relativamente ao primeiro teste e 61% relativamente ao segundo, o que pode ser explicado pelo facto deste repositório não executar leituras. No campo das latências verifica-se que o Cassandra obteve o menor valor de latência, seguido do MongoDB e por último o HBase.

A utilização do MixCloud juntamente com os vários repositórios em simultâneo permite obter melhor desempenho relativamente ao *middleware* com apenas um repositório, concretamente o MidHBase. A melhoria no desempenho advém do facto de as escritas estarem a ser executadas pelo Cassandra, que do conjunto é o primeiro a responder.

Este teste permite comprovar que o Cassandra participa activamente nos testes anteriores na execução das operações de escrita, de forma a contribuir positivamente para o desempenho.

5.4 Workload *read-modify-write*

As aplicações de acesso a sistemas de bases de dados apresentam um padrão de acessos formado maioritariamente por leituras seguidas de escritas, ou seja, é realizada num primeiro passo uma leitura para verificar o estado corrente de uma linha e, consoante a resposta, é lançada uma alteração para essa linha. A estas operações chamamos de operações *read-modify-write*. No contexto do *benchmark*, estas operações realizam uma leitura seguida de uma escrita sobre uma mesma linha, alterando os seus atributos. Nesta base foi criado o *workload read-modify-write* que divide o conjunto de operações em 50% só leituras e 50% do tipo da operação indicado anteriormente. As operações do teste são mapeadas em *Reads* e *Puts*.

Os resultados apresentados nas figuras 5.7 e 5.8 mostram que o desempenho do Cassandra é o pior do teste enquanto o MongoDB obtém a melhor marca, com uma diferença de cerca de 1500 operações/segundo entre eles. O gráfico das latências está consistente com o resultado anterior, onde o Cassandra apresenta o valor mais alto de latência nas operações de leitura e o MongoDB o valor mais baixo. Quanto às operações de escrita, os três sistemas obtiveram um valor de latência semelhante. A latência das operações *read-modify-write* é a soma das latências das operações de escrita e leitura.

O teste aqui realizado assemelha-se ao teste apresentado no workload *write-heavy*. Os resultados dos dois testes variam na mesma proporção para todos os sistemas, ainda que os valores de desempenho obtidos neste teste sejam mais baixos do que no primeiro. Esta

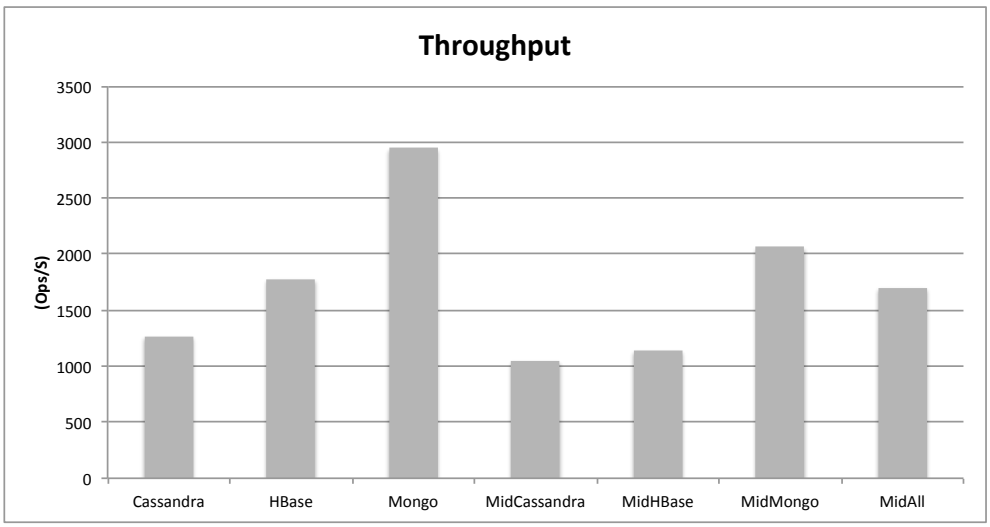

Figura 5.7: Desempenho do sistema (carga com 50% leituras e 50% leituras seguidas de escritas).

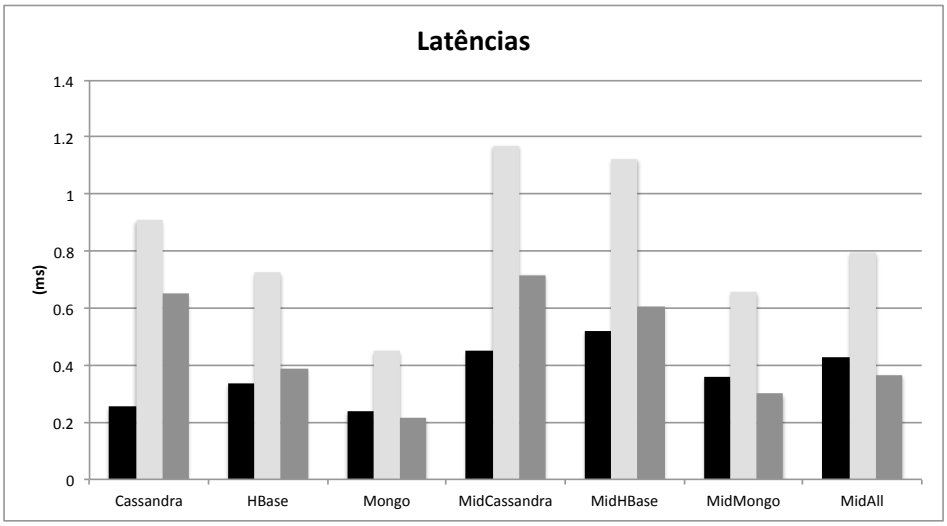

Figura 5.8: Latência das operações (carga com 50% leituras e 50% leituras seguidas de escritas). As barras a preto representam a latência para as operações de escrita, as cinzentas claras para as de *read-modify-write* e as cinzentas escuras para as de leitura.

variação deve-se à introdução da sequência de operações *read-modify-write* traduzindo-se em mais acessos aos repositórios. Adicionalmente, com aumento do número de acessos aumenta o *overhead* das comunicações entre componentes.

Os repositórios apenas com o *middleware* apresentam uma quebra no desempenho em média de 30% e um aumento nas latências das operações. Utilizando o sistema MixCloud obteve-se uma melhoria de desempenho relativamente ao MidCassandra, MixHBase e Cassandra. O aumento de desempenho é explicado pela distribuição das operações pelos repositórios mais rápidos a responder, obtendo resultados próximos do MidCassandra para as escritas e do MidMongo para as leituras. Assim conseguimos criar um sistema que apresenta igual ou melhor desempenho relativamente ao uso de alguns dos repositórios com *middleware*.

5.5 Discussão dos Resultados

Os testes mostram em geral consistência na forma como os sistemas se comportam em diferentes tipos de aplicação. Concretamente, na ausência do MixCloud, o sistema MongoDB apresenta o melhor desempenho para aplicações que realizam um elevado número de leituras. Isto é ser devido ao mecanismo de índices implementado no MongoDB que permite tornar as pesquisas eficientes, enquanto que o Cassandra e o HBase apresentam uma organização dos dados que obriga a uma pesquisa mais exaustiva pelos *tablets*, elevando o número de acessos aos documentos em disco. Por outro lado, o sistema Cassandra destaca-se dos restantes sistemas em aplicações que realizam maioritariamente escritas nos repositórios. O mecanismo de *log* sequencial de escritas em memória presente no Cassandra permite acelerar a execução das operações de escritas. Apesar de

HBase aplicar o mesmo mecanismo que o Cassandra, apresenta pior desempenho em escritas. Uma razão plausível para este resultado é a execução em maior número de passos de operações de escrita. Este caso merece uma análise mais aprofundada que não foi possível realizar durante a elaboração deste trabalho. Estas conclusões são reflectidas no *workload* no qual cada um teve melhor prestação: o MongoDB no teste *workload read-mostly* e o Cassandra e o HBase no *workload write-only*.

Na presença do *middleware* com apenas um repositório, os testes apresentam em geral uma descida no desempenho na ordem dos 40%. Esta descida deve-se à introdução de *overhead* pelo *middleware* causada pela latência da comunicação entre os componentes e implementação do modelo de transformação operacional. Esta última apenas afecta directamente o tempo de execução das operações de escrita. Relativamente à execução das operações de leitura, em geral, do acréscimo da sua latência se espera uma relação de proporcionalidade directa, pois o *middleware* é comum às três implementações (MidMongo, MidHBase e MidCassandra) e assim também os *overheads* respectivos durante a execução. Como é expectável, a latência para ambas as operações em todos os sistemas sofreu um acréscimo de cerca de $20 \sim 40\%$, variando com os sistemas e os testes. O MidMongo e o MidHBase são os sistemas onde a latência aumentou proporcionalmente em relação ao Mongo e ao HBase, respectivamente. No entanto, o MidCassandra apresenta uma subida superior a essa proporção para as leituras. Na origem desta subida está possivelmente um *bottleneck* no servidor MixCloud, resultando num maior tempo de execução para cada operação de leitura.

Com a implementação do *middleware* integrando os três repositórios (MidAll), verifica-se uma melhoria em geral relativamente ao *middleware* com apenas um repositório. Esse facto deve-se à amortização do *overhead* possibilitada pela distribuição dos pedidos pelos vários repositórios (ver, por exemplo, figuras 5.1 e 5.2). Um aproveito da menor latência na execução das leituras no MidMongo comparativamente ao MidCassandra e MidHBase na integração dos três resulta num melhor desempenho global e consistente. A excepção verificada é no teste *workload write-only*, onde o MidAll apenas supera no desempenho do MidHBase. O MidAll apresenta um desempenho melhor para o teste *workload read-mostly* e pior em *workload read-modify-write*. Neste último teste, a integração de uma nova operação constituída por leituras e escritas simultaneamente faz subir a latência numa percentagem superior a cada uma das operações individuais, piorando o seu desempenho.

Estes resultados mostram efectivamente que o MixCloud explora as melhores características de cada um dos sistemas que integra.

Um possível melhoria seria a introdução de um repositório com melhores desempenhos na execução de escritas comparativamente aos presentes, que permitisse compensar o *overhead* e ainda assim obter melhores níveis de desempenho nesta área.

6

Conclusões

No âmbito da dissertação foi proposto e implementado o sistema MixCloud, um sistema que integra diferentes repositórios chave/valor. Nestes sistemas, os dados são replicados num conjunto de repositórios, o que permite fornecer um serviço de armazenamento robusto e com elevada disponibilidade que tolera falhas nos repositórios individuais. Assim, os repositórios usados não necessitam de replicação interna. O MixCloud oferece uma interface que mapeia de forma transparente o modelo de dados e as operações fornecidas sobre as dos diferentes repositórios. Adicionalmente, este sistema explora as diferenças de desempenho dos vários repositórios para fornecer o melhor desempenho.

Os dados são replicados através do envio das operações de escrita para todos os repositórios em simultâneo. Por questões de desempenho, as operações de leitura são apenas enviadas para o repositório mais rápido a responder no momento. O serviço implementa garantias de consistência, utilizando uma aproximação baseada em transformação operacional, permitindo que os servidores executem o conjunto de operações por ordem equivalentes alcançando o mesmo estado final.

Os resultados mostram que, ao adicionar o MixCloud a um repositório individual, o desempenho do serviço tem um decréscimo considerável. Verifica-se a existência de um *overhead* excessivo introduzido pelo *middleware* devido à comunicação entre componentes e ao processo de transformação operacional. No entanto, ao utilizar o MixCloud com os vários repositórios integrados, o *overhead* é amortizado e torna-se possível tirar partido da distribuição dos pedidos por repositório. Na generalidade dos testes, o MixCloud ficou abaixo das expectativas, não conseguindo cumprir com o objectivo de aumento de desempenho do serviço.

Como trabalho futuro pretende-se implementar um mecanismo de comunicação de

difusão de operações entre o cliente e os servidores mais eficiente e desenvolver optimizações no código para diminuir o *overhead*. Adicionalmente, pretende-se criar um mecanismo dinâmico de distribuição de carga que permita distribuir as operações de leitura por diferentes servidores, com base em heurísticas de latência de execução de operações de leitura, por exemplo. Esta aproximação poderá melhorar o desempenho do sistema através da distribuição da carga a leitura. O ambiente de teste deve também ser alargado para conter repositórios replicados a trabalhar no seu ambiente ideal, de forma a testar o impacto real do *middleware* sobre esses sistemas e encontrar soluções para colmatar esse impacto. No campo da tolerância a falhas, queremos propor e implementar um modelo de consenso para tolerância a falhas bizantinas a fim de tornar o serviço robusto e aumentar a disponibilidade e confiança dos dados.

Bibliografia

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, e Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Relatório Técnico UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[BCQ+11] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, e Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pág. 31–46, New York, NY, USA, 2011. ACM.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, e Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[CDK07] G.F. Coulouris, J. Dollimore, e T. Kindberg. *Sistemas Distribuidos: conceitos e projeto*. Bookman Companhia ED, 2007.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, e Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, e Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pág. 143–154, New York, NY, USA, 2010. ACM.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, e Werner Vogels. Dynamo: amazon's highly available key-value

store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pág. 205–220, New York, NY, USA, 2007. ACM.

[EG89] C. A. Ellis e S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18:399–407, June 1989.

[GGL03] Sanjay Ghemawat, Howard Gobioff, e Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pág. 29–43, New York, NY, USA, 2003. ACM.

[Had11] Hadoop file system, Maio 2011. http://hadoop.apache.org/hdfs/.

[HBa11] Hbase system, Agosto 2011. http://hbase.apache.org/.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, e Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pág. 654–663, New York, NY, USA, 1997. ACM.

[Lam78] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[LM10] Avinash Lakshman e Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[Mon11] Mongodb system, Agosto 2011. http://www.mongodb.org/.

[SE98] Chengzheng Sun e Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, CSCW '98, pág. 59–68, New York, NY, USA, 1998. ACM.

[Sha11] Sharding Introduction - MongoDB, Julho 2011. http://www.mongodb.org/display/DOCS/Sharding+Introduction.

[TDP+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, e Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the third international conference on on Parallel and distributed information systems*, PDIS '94, pág. 140–150, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[Zoo11] Zookeeper system, Maio 2011. http://hadoop.apache.org/zookeeper/.